From Warm to Hot Starts: Leveraging Runtimes for the Serverless Era

Joao Carreira
UC Berkeley

Sumer Kohli
UC Berkeley

Rodrigo Bruno
INESC-ID / Técnico, ULisboa

Pedro Fonseca
Purdue University

ABSTRACT

The serverless computing model leverages high-level languages, such as JavaScript and Java, to raise the level of abstraction for cloud programming. However, today's design of serverless computing platforms based on stateless short-lived functions leads to missed opportunities for modern runtimes to optimize serverless functions through techniques such as JIT compilation and code profiling.

In this paper, we show that modern serverless platforms, such as AWS Lambda, do not fully leverage language runtime optimizations. We find that a significant number of function invocations running on warm containers are executed with unoptimized code (*warm-starts*), leading to orders of magnitude performance slowdowns.

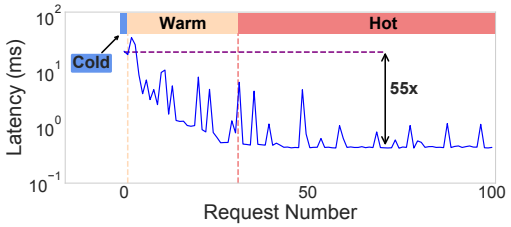
We explore the idea of exploiting the runtime knowledge spread throughout potentially thousands of nodes to profile and optimize code. To that end, we propose Ignite, a serverless platform that orchestrates runtimes across machines to run optimized code from the start (*hot-start*). We present evidence that runtime orchestration has the potential to greatly reduce cost and latency of serverless workloads by running optimized code across thousands of serverless functions.

ACM Reference Format:
Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. 2021. From Warm to Hot Starts: Leveraging Runtimes for the Serverless Era. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31-June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3458336.3465305

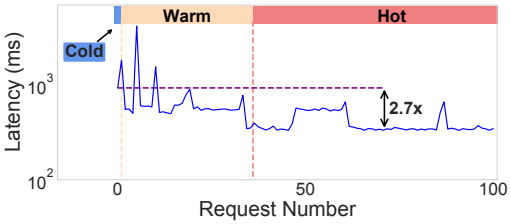
1 INTRODUCTION

Serverless computing is gaining traction as a cloud computing model that promises radically simpler development and deployment of highly distributed applications such as data analytics [3, 4, 19, 22, 27, 30, 31, 33], machine learning [14, 15], code compilation [23], and video processing [8, 20]. In serverless computing platforms, developers deploy individual units of application logic, *serverless functions*, that are triggered by configurable events.

One of the key goals of serverless computing is to free developers from manually managing the underlying resources used by applications. Serverless computing achieves this by leveraging modern languages runtimes, such as JVM and Node.js, to execute functions. This provides a higher-level model for developers, which significantly simplifies time-consuming tasks related to memory management, code compilation, and dependency management.



(a) Java



(b) Javascript

Figure 1: Latency per request for a word count (WordCount) workload on Java and Javascript runtimes for *cold*/*warm*/*hot* stages. These runtimes improve performance by 55x (Java) and 2.7x (JavaScript) through JIT compilation and code profiling. Periodic spikes occur due to garbage collection.

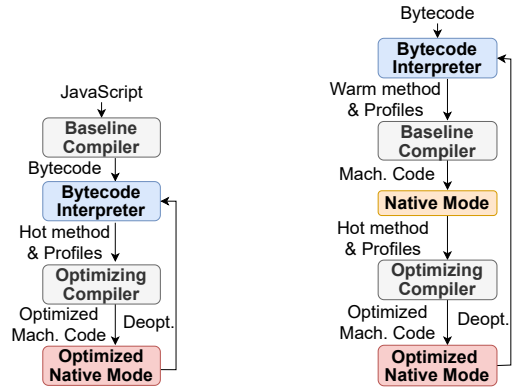
HotOS '21, May 31-June 2, 2021, Ann Arbor, MI, USA

ACM ISBN 978-1-4503-8438-4/21/05.
https://doi.org/10.1145/3458336.3465305

Unfortunately, developers currently face the difficult choice between the simpler development model of serverless and the higher raw compute performance of traditional VM-based platforms. The performance gap between serverless and server-based platforms arises from the lack of co-design between modern language runtimes and serverless computing platforms. Modern runtimes rely on being able to gather code statistics about executions of the same code over time to generate optimized code. However, serverless platforms are tailored for functions that execute sporadically and that take at most a few seconds to complete. The inability to effectively utilize runtime optimizations means that serverless platforms are at a great disadvantage compared to their VM-based counterparts.

Figure 1 demonstrates the drastic impact of runtime optimizations in the performance of serverless functions. In this figure we show the latency of a function that computes a word count workload (WordCount) [16], running on two different runtimes (Correto for Java and Node.js for JavaScript) in AWS Lambda [1]. To isolate the impact of runtime optimizations from the overheads of scheduling and container startup, we ran this function multiple times sequentially inside a single lambda. We observe that the first request is the slowest because the runtime has to load and parse the function code and the function needs to import libraries (*cold*-start). In subsequent function invocations, the runtime can immediately execute the function by interpreting its bytecode and thus we observe an immediate performance improvement after the first couple of invocations (*warm-start*). As the function gets executed multiple times, the runtime generates code profiles that describe what parts of the code are good candidates for code compilation and optimization. Runtimes leverage these profiles and make calls to the Just-In-Time (JIT) compiler to optimize specific functions. We find that all compilations combined result in a 55x (Java) and 2.7x (JavaScript) latency speedup. After the runtime has generated the most optimal code, every new invocation of the function in the same runtime is executed at the maximum performance (*hot-start*).

As our results demonstrate, runtime optimizations can provide significant benefits to serverless functions, provided these functions are executed repeatedly within the same runtime. However most serverless functions run for a short amount of time, making it impossible for runtimes to profile and optimize the function code. A recent study of functions in Microsoft Azure [32] found that on average 50% of functions take at most one second, and 90% take at most 10 seconds. Thus, even when a function invocation is served by a *warm* container, the function may still execute unoptimized code. To make matters worse, after a function is executed, the profiling and compilation information is often discarded and not reused for later executions of the same function. Our



a) JavaScript Engine (V8) b) Java VM (HotSpot)

Figure 2: Code pipeline JavaScript and Java runtimes.

experiments show that these problems lead to up to 72x lower performance for some workloads (see Section 2).

We propose Ignite, a holistic system where runtimes cooperate to generate optimized function code and serverless schedulers can schedule functions depending on the state of the optimization pipeline of each runtime. Ignite reduces profiling and compilation overheads by sharing code optimization information across runtimes. Furthermore, once Ignite has been able to generate optimal code for a specific function, invocations of that function are always *hot* and no profile and compilation overheads are incurred.

2 COLD AND WARM STARTS IN SERVERLESS

In modern serverless platforms each function runs on top of its own independent high-level language runtime (e.g., JVM, Node.js). Within each runtime the function is optimized, through different stages of code transformation, from high-level interpretable code (slow) down to machine code (fast) (see Figure 2). Initially, when a function is invoked within a newly created container, the runtime has to load and parse the code to make it executable by the bytecode interpreter, and initialize its internal datastructures for code execution. Similarly, the function has to load its libraries. Thus, the worst function invocation latency is typically observed at this time (*cold-start*). After this phase, function invocations start executing immediately but execution is not optimal because the function code has not yet been optimized (*warm-start*). For instance, the Java VM Hotspot by default requires thousands of invocations of a function before it gets compiled to machine code and other similar threshold parameters exist for different optimizations. The same happens for the JavaScript V8 engine. Eventually, the runtime identifies and compiles *hot* methods. Subsequent invocations of a function after this stage are executed with optimized code (*hot-start*).

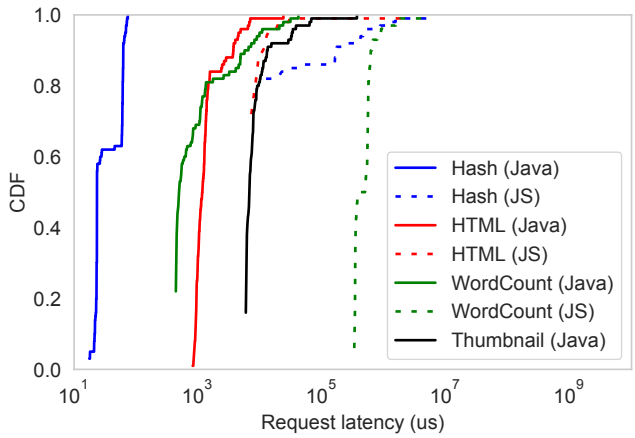


Figure 3: CDF of serverless function invocation latencies for compute-intensive tasks running on top of a Java and JavaScript runtimes.

Even after a function has been optimized by the runtime, the runtime continues to gather code profiling statistics which can lead to new optimizations.

2.1 Performance Benefits of Runtimes

Modern runtimes utilize sophisticated algorithms that can learn how to best improve the performance of an application. For instance, OpenJDK HotSpot JVM, the reference JVM implementation, uses more than 60 compiler optimizations, such as escape analysis, method inlining, and method devirtualization, to generate high performance code during run-time[6]. These optimizations are guided by profiling information that the runtime collects during the execution of the application.

To understand the ability of modern runtimes to optimize the performance of serverless functions, we evaluated the execution of a benchmark suite of JavaScript applications running as functions on top of AWS Lambda, a widely used commercial serverless platform. For this benchmark we selected a set of computationally intensive applications: number hashing (Hash), HTML rendering (HTML), counting words in a text corpus (WordCount), and image thumbnail generation (Thumbnail). We developed each task in Java and JavaScript and created two functions in AWS Lambda, one for each language. All the functions were configured to run with 2 GB of memory. We sequentially invoked each function 100 times and measured how long it took to complete each invocation. Our measurements include only the time spent inside the user function to minimize variance that arises from outside components such as the network.

Figure 3 shows the CDF of the request latencies for all the 4 tasks. For all the tasks in our benchmark, our Java

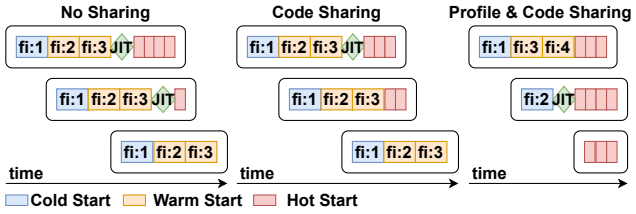


Figure 4: Serverless profile and code sharing modes.

implementation is significantly faster than the corresponding JavaScript implementation. Nonetheless, for all tasks both runtimes are effective at optimizing each function's execution. For instance, we observe that the performance of the tasks when they are *hot* is 3.3-72× (Java) and 3.8-15.2× (JavaScript) faster than when they are *warm*. For some task/runtime combinations, such as Hash (Java) and WordCount (JavaScript), the function performance has two modes: before and after JIT compilation. This can be seen by the single "step" in the CDF. Other task/runtime combinations, such as WordCount (Java) and Hash (JavaScript), are optimized gradually over time and thus their CDFs show several "steps".

2.2 The Waste of Runtime Optimizations

High-level code optimization are highly beneficial for long-running stateful server-based applications. However, unlike typical server-based applications, serverless functions are stateless and short-lived. When serverless containers are recycled to give space to the deployment of other functions, the execution profiles and code generated by runtimes is also discarded. For instance, Shahrad et al. [32] report that Microsoft's Azure Functions [2] recycles serverless containers every 10-20 minutes. Given that 70% of functions are invoked on average less frequently than that, this means that a staggering two thirds of function invocations are executed inside a fresh runtime. Even if platforms keep runtimes in memory between function invocations, at the cost of using extra memory resources, each individual runtime instance still has to go through many *warm* function invocations until code gets *hot*. This problem presents a major obstacle for the adoption of serverless for many workloads.

3 HOT-START FROM THE START!

To overcome this problem, we propose a new runtime-platform co-design, IGNITE, in which serverless platforms are aware of the code compilation pipeline state, and runtimes are able to restore the state of the compilation pipeline upon a function invocation. With this new capability, functions could execute fully optimized code from the start, therefore replacing most *cold-starts* and *warm-starts* with *hot-starts*.

3.1 Holistic Compilation Pipeline

To truly benefit from the progress made locally to profile and compile code, the state of the compilation pipeline must be aggregated and shared. On the one hand, profiles need to be aggregated to increase the amount of information available to guide code compilation. On the other hand, the compiled code needs to be shared to stop further execution of unoptimized code. Figure 4 presents a simple example of how aggregating profiles and sharing compiled code can improve the overall performance of serverless platforms. In this example, a single function is being utilized and invocations are handled in three distinct instances of a language runtime. For simplicity, we consider the entire function code execution mode as cold (first request, requires the initialization of the runtime, in blue), warm (non-optimized code, in orange), and optimized (optimized/compiled code, in red). Inside each non-optimized execution, a single piece of profile information is being accounted, the number of function invocations (*fi*). This profile controls when a function is selected for JIT compilation. For simplicity, we consider the threshold to be three, i.e., a function is compiled after three invocations although in real runtimes this value can be up to thousands. After compilation, functions execute in optimized mode (red), resulting in a 10× speedup compared to non-optimized mode.

Figure 4 presents the execution of multiple function invocations distributed throughout three runtimes. If no sharing is enabled (left), each runtime individually tracks the number of function invocations and compiles the function code. If code sharing is enabled (middle), runtimes profile code locally but once a single runtime compiles the function code, it is shared with other runtimes such that future invocations can benefit from it. If profile and code sharing are enabled (right), runtimes aggregate their profile information and each function invocation is accounted globally. Therefore, once the total invocation counter surpasses the threshold, a single JIT compilation request is issued and the compiled code is shared across all runtimes. In this particular example, for the same workload, a holistic system that takes advantage of the runtime-platform co-design can reduce cost by 1.9×.

3.2 Open Challenges and Opportunities

Reality, however, is more complex than the one presented in the previous example. First, each application method goes individually through the compilation pipeline. Therefore, a single serverless function executes a number of methods which may be at any stage of the compilation pipeline. Second, profiles not only include invocation counters, but also include many other metrics such as conditional branch counters (used for branch prediction [10]), the currently loaded types that implement a method (used to de-virtualize method calls [21]), among others. Finally, the compiled code may target a number of different architectures and may contain a number of assumptions which are only valid locally (such as the availability of particular CPU instruction extensions, or the fact that parts of the code have not been loaded yet). In this section, we discuss the main challenges and opportunities present in this area.

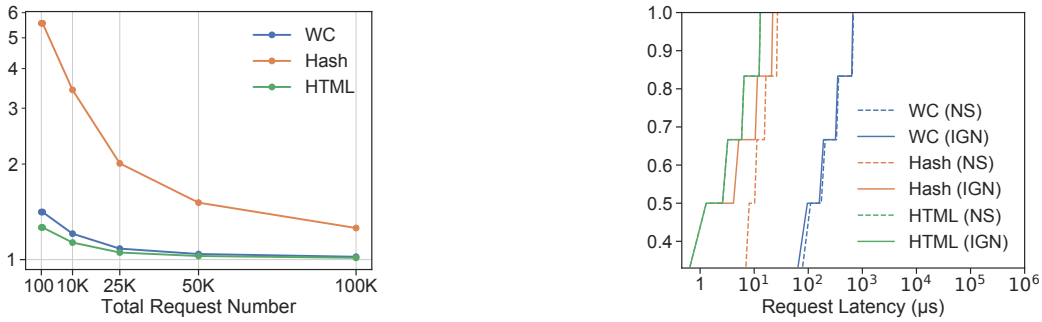
Cross-function Sharing. Sharing profiles and code across functions of different users can be an issue as it would open several attack vectors. For example, if an attacker is able to compromise the runtime's memory and manipulate the profiles that are shared. Even worse, an attacker could alter the code to be shared to bootstrap an attack on runtimes receiving the shared code. Such potential threats mean that sharing profiles and code should be limited to functions of the same user.

Coordination Overhead. A single function invocation can trigger thousands of different methods which will exercise the compilation pipeline. Profiles and compiled code will be produced for each method independently as methods reach their invocation threshold. Sharing all profiles and compiled code for each individual function requires high synchronization overhead but delaying synchronization reduces the number of *hot-starts*. Therefore, a balance between synchronization overhead and global performance must be determined.

Multi-stage Compilation. Compilation pipelines in modern runtimes are organized in stages. Taking HotSpot runtime as an example (see Figure 2), methods can be in any of three stages: interpreted mode, native mode, optimized native mode. Each of these three stages produces different profiling information which needs to be aggregated with profiles produced at the same stage.

Aggressive optimizations. In addition to multi-stage pipelines, compilers are free to apply any aggressive optimization such as loop unrolling, method inlining, among others, based on profiling information. Therefore, each stage of the compilation pipeline is also characterized by the applied set of optimizations.

Dealing with Heterogeneity. A number of compiler optimizations can be subject to local factors. For example, optimizations such as vectorization are only applicable if the CPU supports AVX instructions. Even if the same exact hardware is used, exercising different code-paths will lead to different optimization outcomes in techniques such as method inlining. To successfully aggregate profile and share compiled code, runtimes need to be able to express in which conditions profiles and compiled code are generated such that sharing does not result in incorrect or misguided optimizations.



(a) Cost Reduction Ratio **(b) CDF of Latency**

Figure 5: Simulated improvement of using Ignite (IGN) over a no-sharing (NS) approach in: a) cost (total CPU time), and b) request latency.

Optimizations at Scale. Optimizations are not always successful and sometimes can even degrade performance over time [11]. We envision Ignite being able to leverage the aggregate knowledge of all runtimes running the same function to learn which versions do, and which do not, improve performance.

4 EVALUATION

We now estimate the total cost and latency reduction that can be expected with Ignite in a large deployment. To that end, we developed a serverless platform simulator that simulates the execution of a sequence of invocations of the same function for two types of serverless platforms: one with a no-sharing approach, and one with a profile and code sharing approach (Ignite).

For this simulation we used a set of computationally intensive applications: number hashing (Hash), HTML rendering (HTML), and counting words in a text corpus (WC). We have selected these workloads based on previous benchmarks for serverless platforms [18, 34].

We configured the simulator with function-specific parameters determined experimentally through benchmarks in AWS Lambda. The primary parameters include the request latencies for *cold-starts* (first invocation, includes loading and parsing all function's code), *warm-starts* (already loaded runtime and code, running unoptimized code) and *hot-starts* (container running optimized code), coupled with the number of function invocations necessary to transition between *cold* to *warm* and *warm* to *hot*. When using Ignite, runtimes share profiling data to quickly gather enough information to optimize the code. Such optimized code is then shared with all other runtimes executing the same function.

Results are presented in Figure 5. In this particular simulation, a total of 50 independent containers are simulated and the latencies are estimated based on the measured values from AWS Lambda. Our experiment demonstrates improvements from 1.26× (HTML) up to 5.5× (Hash) for cost reduction and latency reduction compared to no-sharing (see Figure 4). Improvements with IGNITE are higher for workloads for which the JIT compiler can significantly improve performance (e.g., 14.9× latency reduction for Hash). Conversely, for workloads in which the JIT compiler is not as beneficial (e.g., 1.37× for HTML), IGNITE provides a lower cost reduction.

As the number of requests increases, the overall cost reduction ratio converges to 1 because with a high number of requests most function invocations are warm/hot. The simulation considers that the entire platform is running a single function and therefore, no runtime eviction is necessary. In reality, evicting a runtime to execute a different function would further benefit Ignite. Finally, adding more nodes to the platform will just elongate the curves already presented in Figure 5.

5 RELATED WORK

Serverless Optimizations. Recent work, such as snapshotting [13, 17], function co-execution [12, 18], fork-based approaches [7, 29] and warm containers [26], have been proposed to improve the performance of serverless functions. Similarly, runtime reuse has been proposed in the context of serverful platforms to mitigate the overheads of runtime warm-up [24]. All of the proposals for serverless focus on avoiding function startup overheads, such as creating a container and loading the runtime/libraries, and ignore the problem of executing unoptimized code. For instance, snapshotting does not guarantee that the function has been fully optimized at the time of the snapshot. Furthermore, snapshotting significantly increases the storage overhead since every function now requires a full snapshot to be fetched

before execution. Finally, it also raises security issues as runtimes are restored to the same exact state at every function execution, possibly allowing attacks exploiting fixed address layouts. Function co-execution, fork-based, and warm container approaches have been shown to be effective at reducing invocation latency but rely on each runtime to optimize code individually. By doing so, generation of optimized code is delayed. Runtimes individually incur the overhead of code profiling and compilation.

Runtime Optimizations. Ahead-of-time compilation is a viable approach to execute compiled code from the beginning, sidestepping the (potentially) expensive JIT compilation. However, ahead-of-time compilation prevents dynamic code optimizations, such as trace-based inlining, that can generate more optimal code. Profile caching [9] and code caching [5] have also been previously studied, however, separately and in not in the context of Serverless. In Ignite, we aim at combining both techniques and study different scheduling policies that minimize the overhead of profile and code sharing. Last, garbage collection (GC) is another source of runtime overhead that can impact the latency and throughput of serverless applications. Several approaches have been proposed to mitigate the impact of garbage collection in distributed systems [25, 28]. We expect garbage collection to be less of a concern in the context of serverless computing due to the short-lived nature of serverless functions.

6 CONCLUSION

We are implementing Ignite as a new serverless platform to provide hot starts for most function invocations. We aim to leverage the optimizations of several modern runtimes, such as Hotspot JVM and Node.js V8, and intend to experiment with different strategies for sharing profiles and code across nodes.

7 ACKNOWLEDGMENTS

In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware. Rodrigo Bruno's research was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020.

REFERENCES

[1] AWS Lambda. https://aws.amazon.com/lambda/.
[2] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.
[3] Corral. https://github.com/bcongdon/corral.
[4] FaastJS: Serverless batch computing made simple. https://faastjs.org/.
[5] JITServer technology. https://www.eclipse.org/openj9/docs/jitserver/.
[6] JVM JIT-compiler overview. https://cr.openjdk.java.net/~vlivanov/talks/2015_JIT_Overview.pdf.
[7] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. Sand: Towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 923–935, USA, 2018. USENIX Association.
[8] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 263–274, 2018.
[9] M. Arnold, A. Welc, and V. T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 297–311, New York, NY, USA, 2005. Association for Computing Machinery.
[10] T. Ball and J. R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, page 300–313, New York, NY, USA, 1993. Association for Computing Machinery.
[11] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct. 2017.
[12] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky. Putting the "micro" back in microservice. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 645–650, USA, 2018. USENIX Association.
[13] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo. Seuss: Skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.
[14] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, volume 2018, 2018.
[15] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 13–24, New York, NY, USA, 2019. Association for Computing Machinery.
[16] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[17] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.
[18] V. Dukic, R. Bruno, A. Singla, and G. Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 45–59, New York, NY, USA, 2020. Association for Computing Machinery.
[19] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[20] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, 2017. USENIX Association.

[21] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, page 294–310, New York, NY, USA, 2000. Association for Computing Machinery.

[22] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451. ACM, 2017.

[23] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, Oct. 2018. USENIX Association.

[24] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 383–400, Savannah, GA, Nov. 2016. USENIX Association.

[25] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. *SIGPLAN Not.*, 51(4):457–471, Mar. 2016.

[26] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov. Agile cold starts for scalable serverless. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'19, page 21, USA, 2019. USENIX Association.

[27] I. Müller, R. Marroquín, and G. Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. *arXiv preprint arXiv:1912.00937*, 2019.

[28] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 349–365, USA, 2016. USENIX Association.

[29] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Sock: Rapid task provisioning with serverless-optimized containers. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 57–69, USA, 2018. USENIX Association.

[30] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, Feb. 2019. USENIX Association.

[31] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López. Serverless data analytics in the ibm cloud. In *Proceedings of the 19th International Middleware Conference Industry*, pages 1–8.

[32] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[33] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley. numpywren: serverless linear algebra. *arXiv preprint arXiv:1810.09679*, 2018.

[34] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot. *Benchmarking, Analysis, and Optimization of Serverless Function Snapshots*, page 559–572. Association for Computing Machinery, New York, NY, USA, 2021.