Agile Cold Starts for Scalable Serverless

Anup Mohan[1] Harshad Sane[1] Kshitij Doshi[1] Saikrishna Edupuganti[1] Naren Nayak[1]
Vadim Sukhomlinov[2]
Intel Corp.[1], *Google Inc.*[2]

Abstract

The Serverless or Function-as-a-Service (FaaS) model capitalizes on lightweight execution by packaging code and dependencies together for just-in-time dispatch. Often a container environment has to be set up afresh– a condition called "cold start", and in such cases, performance suffers and overheads mount, both deteriorating rapidly under high concurrency. Caching and reusing previously employed containers ties up memory and risks information leakage. Latency for cold starts is frequently due to work and wait-times in setting up various dependencies – such as in initializing networking elements. This paper proposes a solution that pre-crafts such resources and then dynamically reassociates them with baseline containers. Applied to networking, this approach demonstrates an order of magnitude gain in cold starts, negligible memory consumption, and flat startup time under rising concurrency.

1 Introduction

Containerization decouples developers from provisioning of hosting environments, simplifies software solutions, and facilitates frictionless evolution and delivery of services. In comparison with virtual machines (VMs), containers are resource efficient means of performing quick executions, and thus provide for highly responsive on-demand staging of solutions at an exceptional scale [17], –notably, in micro-services oriented and server-agnostic deployments. In particular, the serverless model (also known as Function-as-a-Service or FaaS) permits the developer to focus exclusively on the application logic [13], while freeing up the cloud service provider (CSP) to maximize productive utilization of infrastructure. FaaS has emerged as the natural choice for cloud- and container-based fulfillment [2] [7] [5] [8] as on-demand computation increasingly materializes as event-triggered tasks that arise unpredictably and then run to completion. FaaS introduces new software management considerations that CSPs must weigh [12,15]; these include, security, determinism, and startup overheads of container-based dispatch.

A function's execution must be preceded by the bring-up of its execution environment –which frequently comprises a "cold start" wherein the container is started from scratch. Even though containers are several factors faster than VMs their cold start time can still be a significant fraction of that of a typical function's execution and rise sharply with increased concurrency of function triggers [6] as described in section 3. Concurrent function executions are common especially during rush hours (e.g. Uber) and with function chaining. Common workarounds to reduce cold starts tend to be costly and resource consuming, as explained in section 2.

A detailed analysis of time spent in various stages of a cold start in section 4 identifies network creation and initialization as the prime contributor to latency (also shown in [16]), a common stage involved in almost all the serverless frameworks. The solution proposed in this paper (and prototyped with Apache OpenWhisk) is to (i) pre-create and cache networking endpoints, (ii) bind them to function containers when created, and (iii) salvage them for reuse when function containers are dismantled. This is achieved by using Pause containers [1], which are network-ready empty containers readily attachable to other containers that execute using its predefined network configuration. Our solution includes a management system around this concept, called as Pause Container Pool Manager, and abbreviated as PCPM, reduces cold start latencies and their concurrency impact by nearly an order of magnitude; for example, at hundred concurrent containers, the startup time reduces by close to 80%, while consuming a negligible amount of system memory. Our solution, described in section 5 and evaluated in section 6 is not limited to Openwhisk. Section 7 captures continuing and future work and concludes. Even though this paper focusses on pre-creating network resources, the idea of identifying FaaS performance bottlenecks and pre-creating or pre-fetching resources can be extended beyond network endpoints as discussed in section 7.

2 Related Work

The existing approaches revolve around ensuring that the environment for a function (runtime, dependencies, and versions,

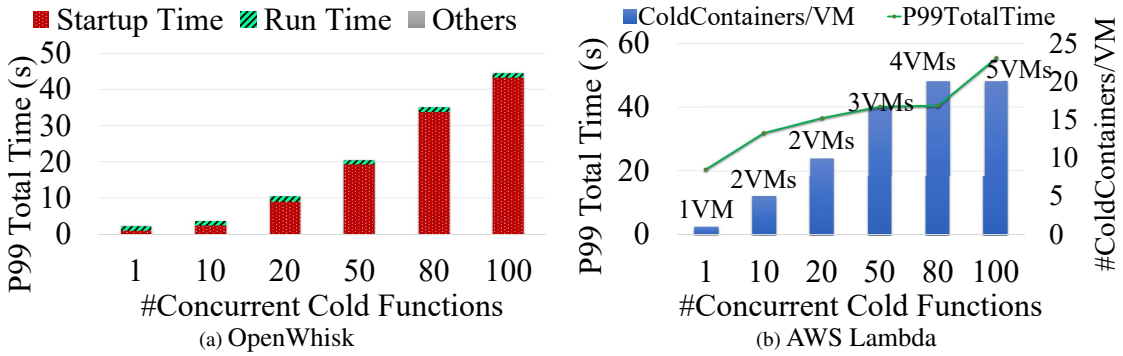


Figure 1: Impact of concurrent cold starts: (a) shows the total time for different concurrent cold starts; total time increases with concurrency, (b) shows the total time under concurrent cold start for AWS Lambda. Total time increases with concurrency and Lambda employs additional virtual machines (VMs) to mitigate this problem. Multiple instances of the same function are used.

known memory and network requirements) pre-arranged for its invocation. One popular approach is *pre-warmed containers* [10] which saves the cost of launching a new container by pre-creating the container with the needed code and its dependencies and starting it up when invoked. A related approach, *Warm containers* [3, 10], goes further and saves the startup time by caching a running copy of the container with the needed environment (between successive executions) and usually this approach ties up more resources. Both techniques are limited to using the containers for the specific environments for which they are created; and failure to do so means either squandering memory, swap space, etc. by overprovisioning such resources, or, degrading scalability and predictability of response time by undersubscribing them. Even slight under-provisioning can provoke cold-start bursts when arrival patterns shift suddenly, and produce large startup times. A heuristic, *periodic warming* [4] consists of submitting dummy requests every so often to induce a CSP to keep containers warm. Non-deterministic and concurrent invocations are not effectively mitigated by this technique. Akkus et al. [11] reduces container counts by running multiple functions from same user inside a container, which is still prone to cold starts as user counts rise.

3 Impact of Concurrent Cold Starts

This section describes the cold start issue and its measured impact in detail. First, we show the cold start symptoms through experimentation using the open source Apache OpenWhisk [8] framework. Here we trigger incremental concurrent cold starts and use custom instrumentation to generate a latency breakdown of execution. Then, we confirm similar effects (but different magnitudes) on AWS Lambda [2]. For both frameworks, we employ a simple, short running, ALU intensive function (i.e. light on caches and memory bandwidth) as the test case; this function calculates prime numbers within a fixed range using the sieve method [9]. Note that the cold start issue is independent of the function and related to the container start up required to run the function.

3.1 Apache OpenWhisk

Our Apache OpenWhisk setup uses a single Intel Xeon® Platinum 8180 server to both host the framework and execute functions. Since OpenWhisk is built upon Docker, we have used Docker's logging and eventing capabilities to trace the life of a function and generate a latency distribution. In Fig. 1(a), we plot the 99^{th} percentile time attributed to various phases under different numbers of concurrent cold starts[1]. For simplicity in Fig. 1(a), we divide the vertical bars into three parts: (i) *startup time* – which is the wall-clock time to create and initialize the Docker container, (ii) *run time*, which is the elapsed time to the function (i.e. compute prime numbers), and (iii) *remaining time* that includes the framework's management overheads such as identifying and meeting various module dependencies, load balancing, user authentication etc. The total time to execute functions increases dramatically with concurrency even though the server has sufficient cores to run these activities independently. More than 90% of this total time is spent in the start up time (shown in red), indicating a scaling bottleneck. The issue is not with the OpenWhisk architecture but with the container network setup in the kernel for the containers, as explained in Section 4.

3.2 AWS Lambda

To ensure we are not merely confronting some Apache OpenWhisk artifact, we also characterized execution on AWS Lambda. Fig. 1 (b) shows the total time to execute Lambda functions for concurrent function triggers. We use the approach described by Wang et al. [18] to determine when new Virtual Machines (VMs) and containers are used by Lambda to execute functions. Fig. 1(b) shows that the total 99^{th} percentile time increases with concurrency, though at a gradual

[1]We cross-verify the startup time measurements by using Docker logs generated by the Docker daemon

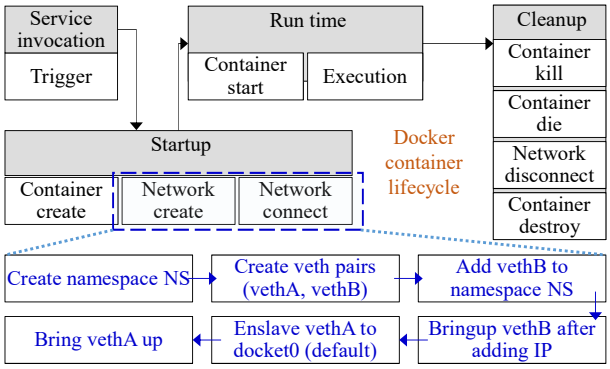


Figure 2: Container and network namespace creation.

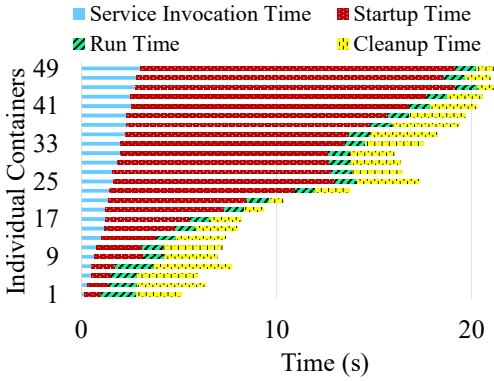


Figure 3: Timeline of 50 concurrent functions with alternate containers. Startup time accounts for 90% of total time.

rate with the aid of multiple VMs used for concurrent function execution. The total time however increases significantly with increasing cold containers within the same VM. As reported in Wang et al. [18], the VMs used for Lambda are approximately 3 GB in size; thus, Lambda incurs significant memory outlay to mitigate the concurrent cold start scaling bottleneck.

4 Role of Network in Cold Starts

Fig. 2, left, depicts the lifecycle of a Docker container (however, the illustration also applies to other containers). It shows that a container goes through four major stages, viz. (i) *service invocation*, in which a function trigger reaches the Docker daemon, (ii) *startup*, which consists of creating a container, setting up its network, and connecting it to the network, (iii) *run time*, in which the function is executed, (iv) *cleanup*, which includes stopping the container, disconnecting its network, and destroying it. The service invocation, startup, and run time fall directly in the critical path of the function execution. The cleanup step could fall indirectly in the critical path of other functions as it demands cycles from Docker daemon.

To understand the scaling bottleneck, we arbitrarily pick the 50 concurrent execution case from Fig. 1 (a) and break down the total execution time of each function into the four steps listed above. Fig. 3 shows a Gantt chart of their execution timelines, which highlights the growth in startup time as the major factor limiting the scaling of performance. The increase in the comparatively modest service invocation time is from the Docker daemon taking more time to process the concurrent requests. The run time remains steady across all the containers. The cleanup time is the second highest contributor to the full execution time. The service invocation, startup, and cleanup are overheads for the FaaS provider.

Referring back to Fig. 2, the startup time is further broken down into Container Create, Network Create, and Network Connect times with the help of the Docker events tracing mechanism. Our analysis finds that the Network Create and Network Connect steps account for 90% of the startup time. To drill further into these network tasks we emulate network

Table 1: Network Namespace Creation and Removal Bottleneck under Concurrency

#Concurrent Namespaces	1	10	50	100
Create Time (s)	0.28	1.27	6.28	14.41
Cleanup Time (s)	0.20	0.71	3.24	7.77

namespace creation within the Linux kernel – by using the Linux *ip* command to create and initialize the network in a manner similar to Docker. The steps in Fig. 2, right, include creating a new network namespace, creating virtual Ethernet (veth) pairs, assigning IP address, and enabling the connections. We create and initialize multiple networks concurrently to emulate the launching of multiple Docker containers. Table. 1 shows that the time to *create network namespaces and initialize them* increases with concurrency, and are the main reason for the super-linear increase in startup time. The cleaning up of a container includes deleting the namespace and veth pairs, and Table 1 shows that this too increases significantly with concurrency. A recent study [16] corroborates our findings and identifies that the scalability bottleneck of namespace creation is due to a single global lock. They [16] also explain that namespace cleaning happens in batches and is less time consuming compared to namespace creation. Virtually all serverless frameworks require creating network namespaces frequently, thereby facing performance hurdle as of Linux kernel version 4.4.0-116 (Ubuntu).

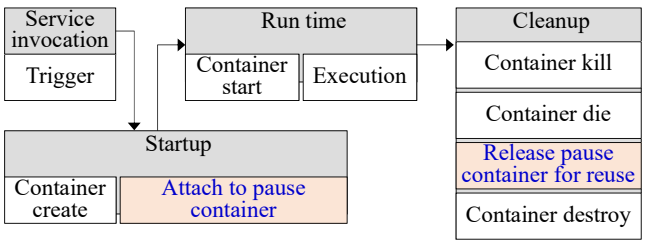
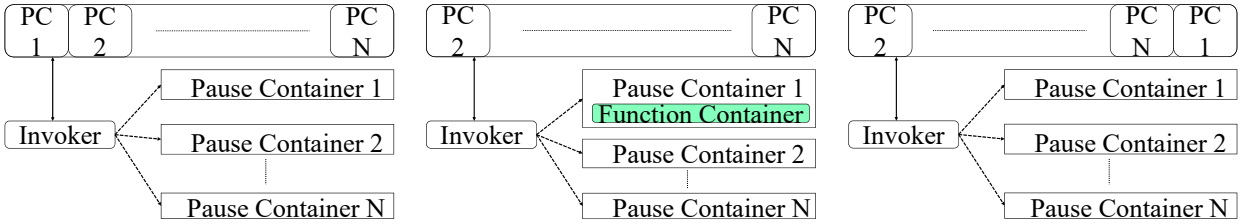


Figure 4: Container lifecycle under pause containers – bypasses time-consuming Network create and connect steps.



(a) Build Phase (b) Execution Phase (c) Completion Phase

Figure 5: Different phases of Pause Container Pool Manager (PCPM)

5 Pause Container Pool Manager (PCPM)

This section describes the approach for the solution this paper takes. It consists of pre-creating networks and connecting them to function containers, so that the time otherwise spent in the network create and network connect steps (Fig. 2) are removed from the critical step of startup. For pre-creation and initialization of networks, we borrow from Kubernetes [14], the concept of Pause containers [1] and use it as explained in Section 5.1. We also develop a pool manager for connecting the appropriate function to a pause container and managing its lifecycle. We prototype the complete solution, termed PCPM, for evaluation under Apache OpenWhisk.

5.1 Network Pre-creation

A pause container (PC) is created ahead of time, with its initialization paused after the network creation step where an IP address is assigned to it. A normal Docker (or any) container is subsequently attached to a PC when needed, thus effectively sharing its pre-established network namespace. Since PCs are part of the (Docker) network, any container attached to a PC is also part of the same network and can communicate with other containers that are part of that network.

Our solution incorporates a pool management system — *pause container pool manager* (PCPM). The PCPM creates PCs and places them in a pool. When needed, they are then taken from the pool and bound to function containers; and when those application containers are ready to terminate, the PCs are detached and placed back in the pool. Fig. 4 shows the resulting new cold start process — now, the time-consuming network-create and network-connect steps of Fig. 2 are replaced by the simple step of just attaching the newly created execution container to a PC taken from the pool.

When the function container terminates, the PC is simply detached and put back in the PC pool for reuse. The PCs are environment agnostic and can be used by any application container; thus, a PC used at one time by a Python function can be later re-used for a Node.js function. Further, except for the network ID, the PCs are stateless, –needing just a few kilobytes each; and the PCs do not become conduits of any information between a previous attachment and a new one.

5.2 Pool Management

As just recounted, the PCPM manages a pool of PCs; it does so in three key phases. The first phase is the build phase during which the FaaS framework is setup and initialized. Fig. 5 (a) shows the build phase of a generic container based framework. The module responsible for launching containers and executing tasks is referred to as *Invoker* in OpenWhisk. As shown in Fig. 5 (a), during the build phase, a number of pause containers are launched. At this time, PCPM initializes a free pool with the identifiers corresponding to the PCs. The invoker has knowledge about the PCs through the PCPM.

Fig. 5 (b) shows the second phase, – namely the execution phase. During this phase, the invoker queries the pool manager and obtains the identifier of an available PC (e.g., PC 1 in Fig. 5 (b)). Using this identifier, the invoker attaches the newly launched container to the corresponding PC (i.e., PC 1), and removes it from the free pool.

Fig. 5 (c) depicts the third phase, –namely the completion phase, when a function is completed and its execution container is terminated or otherwise recycled. At this point, the invoker contacts the pool manager and the pool manager reclaims PC1 and inserts the identifier back into the free pool. The point of insertion doesn't matter as PCs have no data retention and therefore, no locality benefits.

It is beneficial to keep a large number of PCs in the pool to support function execution at scale, and this is reasonable given their small memory footprint. The Linux kernel has a limit on creating multiple veth pairs on a single network bridge (1024) but this is easily mitigated by creating multiple network bridges and linking them together to support a large number ($>>$ 1024) of PCs. In the absence of free PCs, a default startup process will follow. A requirement for using PCs is that the network configuration of the function container match that of the PC, and since in OpenWhisk, function containers are expected to have port 8080 exposed, the PCs expose port 8080 as well. Even though non-standard network requirements are unlikely since common FaaS frameworks may not support them, it is easy to accommodate them when they arise, by creating multiple variety of PC pools for them under PCPM. Even though we built PCPM for evaluation within OpenWhisk, it can be implemented for any other FaaS framework, as the basic concept of pre-creating and initializ-

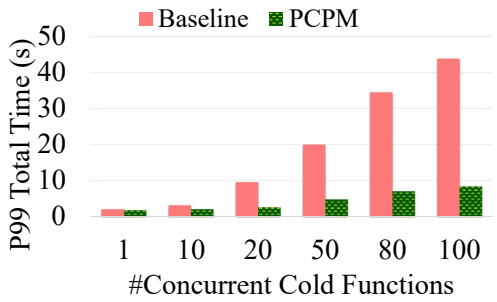


Figure 6: Reduction in cold start time with PCPM. PCPM reduces the cold start execution time up to 80%.

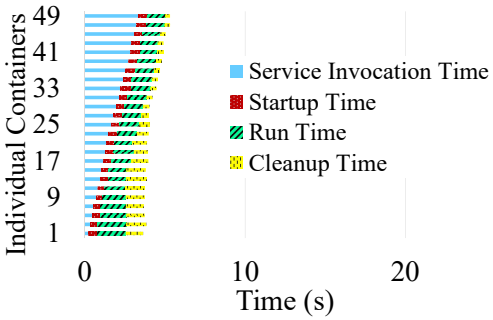


Figure 7: Timeline for 50 concurrent functions with PCPM with alternate container shown (cf. Fig. 3).

ing networks is common across frameworks.

6 Results

We evaluate concurrent executions for our PCPM-based OpenWhisk using unmodified OpenWhisk for baseline. In this section we compare with the pre-warm and warm container mitigation and show up to 80% savings in execution time (relative to cold starts) and memory saving of several orders of magnitude relative to pre-warm/warm containers.

6.1 Comparison with OpenWhisk

Fig. 6 compares its execution time with that of baseline, for concurrency levels ranging from one to hundred. At fifty concurrent cold starts, execution time is cut by 75%; this improves further to a cut of 80% at hundred concurrent cold starts. Fig. 7 shows a Gantt-chart for fifty cold-start executions under PCPM-OpenWhisk for comparison with the baseline in Fig. 3 (a). It may be noted that the cleanup time is also reduced as PCPM avoids deleting network namespaces and veth pairs during terminations. Work is in progress to identify next level bottlenecks and reduce the service invocation time.

6.2 Comparison with Pre-warm and Warm

Table 2: PCPM vs Cold, Pre-warm, and Warm Containers

Container Type	Environment Dependency	Function Dependency	Mem Usage (MB) (50 Containers)	Total Time (s) (50 Functions)
Cold	NO	NO	0	20.01
Warm	YES	YES	1600	0.78
Pre-Warm	YES	NO	1500	1.05
PCPM	**NO**	**NO**	**2**	**4.84**

Table 2 compares PCPM-based cold-starts with baseline executions that are pre-warmed, warmed, or neither. Cold container execution has no memory pre-committed (column 4). Warm execution caches containers and code dependencies, and thus has the highest memory footprint. The pre-warm case has the execution environment (e.g. Python) ready but must bring in the code dependencies, causing a high footprint. The memory footprint for the PCPM-based execution is negligible. For our simple prime number function, the footprint and time of warm container compare well with those of pre-warm containers. The gap between PCPM and Pre-warmed time, while much smaller than cold-start, reflects the dominance of the process setup work over the actual work in the function itself. However, for more complex work (e.g., neural network inference), one should expect significantly larger memory footprints but modest time reduction for warm containers over pre-warm containers. Similarly the gap in memory footprint between PCPM-based and the pre-warm or warm containers can be expected to widen significantly. For such complex actions, the relatively small contribution of the process setup should make the total time comparable between PCPM-based and pre-warmed activations.

7 Summary and Discussion

This paper presented the ideas of pre-creating resources to improve the FaaS performance. We focused on the container cold start issue and identified the network creation and initialization to be the major reason for the performance bottleneck. The proposed solution pre-creates networks (instead of pre-creating containers), and seamlessly attaches the pre-created networks to function containers. To pre-create networks, this paper uses pause containers, and to govern their lifecycle, it creates a pause-container pool manager (PCPM) system. Evaluation (using an OpenWhisk based prototype) demonstrates up to 80% reduction in execution time compared with cold containers, and several orders of magnitude reduction in memory footprint with a modest drop in performance compared with pre-warmed containers. This solution does not exclude the use of current approaches in which function containers may themselves be pre-created and/or cached for reuse; however, it makes it significantly less necessary to adopt such measures and tie down memory or clamp trigger rates.

With PCPM re-using pause containers and thereby IP addresses, we would like to add multiple security and isolation policies over network namespaces to strengthen the security aspects of PCPM. The idea of pre-creating or pre-fetching resources can be extended beyond network entities. Some opportunities include pre-creating components of function environments and pre-fetching code dependencies with the help of the container orchestrator to make FaaS agile with minimal memory footprint.

References

[1] The almighty pause container. https://www.ianlewis.org/en/almighty-pause-container.

[2] AWS Lambda. http://aws.amazon.com/lambda/.

[3] Cold start / Warm start with AWS Lambda. https://blog.octo.com/en/cold-start-warm-start-with-aws-lambda/.

[4] Dealing with cold starts in AWS Lambda. https://medium.com/thundra/dealing-with-cold-starts-in-aws-lambda-a5e3aa8f532.

[5] Google Cloud Functions. https://cloud.google.com/functions/.

[6] I'm afraid you're thinking about AWS Lambda cold starts all wrong. https://hackernoon.com/im-afraid-you-re-thinking-about-aws-lambda-cold-starts-all-wrong-7d907f278a4f.

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

[8] OpenWhisk. https://github.com/apache/incubator-openwhisk.

[9] Sieve of eratosthenes. https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes.

[10] Squeezing the milliseconds: How to make serverless platforms blazing fast! https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0.

[11] Istemi Ekin Akkus et al. SAND: Towards high-performance serverless computing. In *USENIX Annual Technical Conference)*, pages 923–935, 2018.

[12] Ioana Baldini et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[13] Rajkumar Buyya et al. A manifesto for future generation cloud computing: Research directions for the next decade. *ACM Computing Surveys (CSUR)*, 51(5):105, 2018.

[14] Inc Kubernetes. Kubernetes-production-grade container orchestration. https://kubernetes.io/.

[15] Wes Lloyd et al. Serverless computing: An investigation of factors influencing microservice performance. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169. IEEE, 2018.

[16] Edward Oakes et al. SOCK: Rapid task provisioning with serverless-optimized containers. In *USENIX Annual Technical Conference*, pages 57–70, 2018.

[17] Stephen Soltesz et al. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[18] Liang Wang et al. Peeking behind the curtains of serverless platforms. In *USENIX Annual Technical Conference)*, pages 133–146, 2018.