Photons: Lambdas on a diet

Vojislav Dukic, Rodrigo Bruno, Ankit Singla, Gustavo Alonso
Systems Group
Dept. of Computer Science
ETH Zürich

Abstract

Serverless computing allows users to create short, stateless functions and invoke hundreds of them concurrently to tackle massively parallel workloads. We observe that even though most of the footprint of a serverless function is fixed across its invocations — language runtime, libraries, and other application state — today's serverless platforms do not exploit this redundancy. Such an inefficiency has cascading negative impacts: longer startup times, lower throughput, higher latency, and higher cost. To mitigate these problems, we have built *Photons*, a framework leveraging workload parallelism to co-locate multiple instances of the *same* function within the same runtime. Concurrent invocations can then share the runtime and application state transparently, without compromising execution safety. *Photons* reduce function's memory consumption by 25% to 98% per invocation, with no performance degradation compared to today's serverless platforms. We also show that our approach can reduce the overall memory utilization by 30%, and the total number of cold starts by 52%.

CCS Concepts

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → *Virtual machines.*

Keywords

serverless computing, shared runtime, workload collocation

ACM Reference Format:
Vojislav Dukic, Rodrigo Bruno, Ankit Singla, Gustavo Alonso. 2020. *Photons*: Lambdas on a diet. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421297

1 Introduction

Serverless computing promises massive parallelism at low cost by executing functions (tasks) in lightweight containers instead of conventional, longer-term resource allocation with virtual machines. Serverless functions are short-lived, stateless operations programmed in easy-to-deploy, high-level languages such as JavaScript, Python, or Java.

Despite the name, serverless functions do use a server. When a function is invoked by an event, e.g., an HTTP request arrival or a timer trigger, the cloud provider allocates a container with a predefined amount of CPU and memory resources. After resource allocation, the container initializes a runtime (*e.g.,* a Python runtime or a Java Virtual Machine) with all the necessary application-specific libraries, code, and data needed to process the event.

By avoiding static resource allocation in large chunks (like VMs) and being able to scale quickly, serverless platforms achieve high performance and low overall cost for bursty event-driven applications, like Web workloads, or those that exploit massive parallelism, like data analytics. The nature of these workloads often results in numerous concurrent executions of the *same* function [54]. Although serverless computing provides a significant efficiency leap forward compared to more traditional platforms, there is still substantial room for improvement.

Today's serverless platforms initialize and schedule each invocation separately [18, 22], even when numerous invocations execute the same code and need the same environment. This strict isolation causes two major inefficiencies: (a) there is no memory sharing across invocations, which increases the overall memory utilization; and (b) each invocation has to initialize its own runtime and application state, which prolongs the execution time.

We argue that this strict isolation is not essential for the safe *concurrent execution of the same function.* Since parallel invocations of the same function can trust each other, they can be collocated and share an execution context, ***if*** we can: (a) ensure safe execution by separating their state / data from each other; and (b) scale resources appropriately such that different concurrent invocations suffer minimal or no resource contention.

Leveraging this observation, we present *Photons*: an ultra-lightweight execution context based on runtime and app-state virtualization for serverless functions.[1] *Photons* provide the same serverless abstractions as today's platforms while allowing safe runtime sharing for applications that use the large scale parallelism of serverless to concurrently run many instances of the same operation. *Photons* leverage runtime-level isolation to provide automatic data separation for collocated invocations of the same function, while still enforcing strict memory isolation across different functions using existing virtualization technology like Docker [48] or Firecracker [22]. For a Java Virtual Machine (JVM) target[2], we present an implementation that automatically separates the private state of collocated invocations and allows transparent sharing of the shareable application state. We discuss how the same approach could be easily implemented for other popular runtimes, such as those of Python and JavaScript. We also show how, by determining the invocation-specific resource footprint of a function, resources can be appropriately provisioned such that concurrent executions sharing an environment do not suffer from resource contention.

We implement *photons* in OpenWhisk, an open-source serverless platform, and demonstrate their benefits.[3] Our experiments (§5) show that *photons* reduce memory consumption by 25% to 98% per invocation for common serverless workloads. Using simulations with traces from the Microsoft Azure serverless infrastructure [54], we demonstrate that *photons* can reduce the overall cluster memory utilization by 30%, while reducing the total number of (slow) cold starts by 52%. All of our results account for the small overheads we incur to enforce data separation for collocated invocations.

2 Motivation

We identify the inefficiencies in today's serverless platforms when invoking the same function concurrently. We then quantify the extent to which sharing the execution context across such invocations can overcome these inefficiencies. Finally, we provide evidence of the widespread use of workloads that can benefit from context sharing.

2.1 Inefficiencies in today's platforms

The first generation of serverless computing currently deployed by major cloud providers came with many inefficiencies [39]. Recognizing the potential of these platforms, many proposals extend serverless computing by addressing questions such as how to communicate among serverless functions [19, 43], how to deploy data analytics workloads on serverless platforms [50, 53], and even how to build new serverless platforms [2, 14, 16]. Our work complements these efforts and the other related work in this exciting area (§3) by focusing on the following inefficiencies of serverless infrastructures:

Runtime memory overhead: For a serverless function with a small amount of invocation-specific state, *e.g.,* a REST API call, an entire runtime of tens of MB is loaded to process a few KB of invocation data. Each concurrent invocation uses its own copy of the entire execution context and incurs a large overhead loading the runtime.

Redundant data: Consider a machine learning inference application processing multiple inputs in parallel. Each inference request requires the pre-trained model, often hundreds of MB, to be fetched from storage. A lack of model sharing across concurrent invocations introduces substantial overhead in terms of function execution time, memory use, and network traffic for the necessary I/O.

Startup delay: Loading a fresh runtime for each invocation introduces a startup delay. This "cold start" can last from 100 ms to a couple of seconds on today's platforms [34, 55]. Even the fastest function startup solutions prevent many low-latency workloads from being deployed atop serverless infrastructure because the startup overhead is significantly larger compared to useful work (< 10 ms) per invocation.

Poor multiplexing: Although serverless platforms present a significant leap forward compared to VMs in terms of resource utilization, they still suffer from a similar problem — each invocation's allocation of CPU and memory resources is fixed during its execution. For functions performing significant I/O, being able to interleave concurrent invocations across shared resources would improve resource utilization even further and reduce cost.

2.2 Context sharing opportunity

We illustrate the value of context sharing by analyzing an image classification task[4] running on top of Amazon's Lambdas. Each function fetches an image from Amazon S3 cloud storage and uses TensorFlow running in either Java or Python to classify it (details in §5). Figure 1 shows the memory consumption of each invocation. The runtime, libraries, and application state (the machine learning model) use most of the memory, with only 6% and 29% of the memory being specific to the invocation for Java and Python, respectively. In this application, everything can be shared apart from the invocation-specific state.

[1]A photon is a mass-less, light-speed particle, hence the name.

[2]We chose the JVM because it is a complex target that supports multiple languages — Java, Scala, Kotlin, and Groovy, among others.

[3]To support future work, we release our source code: https://github.com/rodrigo-bruno/openwhisk-runtime-java.

[4]There is broad interest in running such workloads on serverless platforms: image classification is the "hello world" task of serverless machine learning, used in serverless tutorials by each of Amazon AWS [10], Microsoft Azure [3], and Google GCP [11].

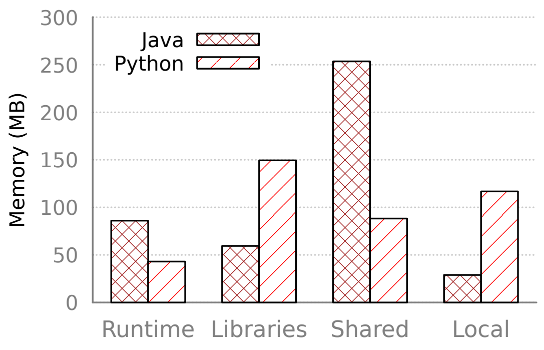


Figure 1: The breakdown of memory use for a serverless image classification task: most of the memory is used for the runtime, libraries, and the machine learning model, which remain the same for all invocations.

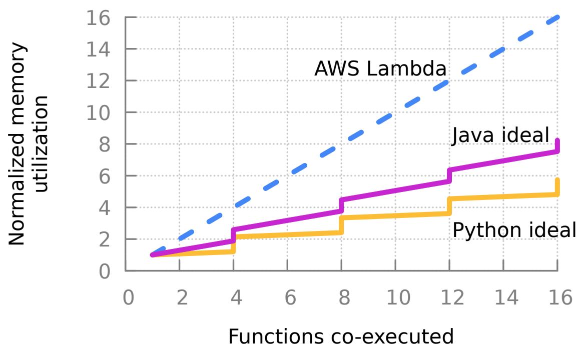


Figure 2: Today's platforms (AWS Lambda here) allocate each instance of a function in its own container. If we were to exploit the redundancy shown in Figure 1 by sharing the execution context across a number of concurrent functions (C = 4 here), memory utilization would be lower.

This further means that if concurrent executions are collocated on the same machine, they have to allocate the shareable context only once. To avoid resource contention, only a limited number of invocations, C, should share an execution context. The concurrency parameter C is defined as $\left\lfloor \frac{c_{av}}{c_{invo}} \right\rfloor$, c_{av} is the amount of resources available on a particular machine, and where c_{invo} is the resource requirements per invocation. Collocating more than C invocations require a new execution context to be allocated on another machine. Note that C depends on the amount of resources available and can change over time. Such sharing would provide substantial memory savings, as in Figure 2, where we show results for the increasing number of concurrent image classification invocations n, with $C = 4$. The results are normalized to the memory usage of one AWS Lambda invocation. Ideally, every set of C concurrent invocations would share all shareable state, using new memory only for private invocation state. Asymptotically, today's serverless frameworks use 3.3× and 2.1× more memory than necessary for the Java and Python implementations, respectively.

Besides the memory footprint reduction, collocated invocations would see an improvement in execution time as well, since the shared context has to be created only for the first invocation, and is available immediately to the other $C - 1$ invocations, avoiding the cold start.

2.3 The limits of context sharing

The degree to which a particular function can benefit from context sharing depends on what fraction of its memory use is invocation specific. If a function's memory use consists of P units of invocation-specific memory and S units of shareable memory ($\frac{P}{S} = \frac{6}{94}$ for the Java classifier in Fig. 1), then the memory-use efficiency (δ) is defined as:

$$\delta = \frac{Memory(AWS\ Lambda)}{Memory(Ideal)} = \frac{P + S}{P + S/C}$$

Note here the obvious similarities to Amdahl's law [24] — in essence, we are doing for memory-use what partially-parallel processing does for compute. We make three observations from the above expression:

- δ does not depend on system load, but only on the proportion of private and shareable memory, and the maximum number of invocations per shared context, C.
- If private memory dominates the memory consumption, $\delta \longrightarrow 1$, implying no benefit from shared context.
- If shared memory dominates, $\delta \longrightarrow C$, implying a large, $C\times$ memory efficiency improvement.

2.4 Concurrent invocations are common

Context sharing is only possible if there are multiple invocations of the *same* function executing at the same time. However, this is the case for many serverless workloads.

Due to its use of short, ephemeral functions, and the possibility of invoking a large number of these in parallel, serverless computing is being used for both event-driven workloads, where a VM would be idle most of the time, and for massively parallel, short-term compute, like in data analytics. Example use cases include Internet of Things and edge computing [33], parallel data processing [41], video processing [35], and Web backend APIs [6]. A common property of these workloads is that they are bursty — they make a large number of invocations of a few functions to either tackle bursts in event-driven workloads or exploit data parallelism for analytics.

The ubiquity of invocation concurrency is concretely captured in a recent study of serverless workloads on the Microsoft Azure production serverless platform [54]. The study shows that 1% of all functions is responsible for more than 90% of all invocations, and as we show later (§5), the observed invocation arrival frequency, execution time distribution, and overall cluster load, give sufficient opportunity for highly effective context sharing.

3 Design space and related work

Today's serverless platforms use specialized container systems [9, 13, 22] to cut startup times to around 100 ms. While this is a substantial improvement over traditional virtualization using, *e.g.,* KVM or QEMU, this long startup time is still prohibitive for latency-sensitive functions. Further, even the new container technologies do not ameliorate the drawbacks discussed in §2.1. We discuss several proposed solutions to such issues.

Warm containers: On today's platforms, after an invocation finishes, its container is not immediately destroyed, but rather kept in memory until a timeout occurs, or the memory is needed for another container [1, 20, 56]. This enables a future invocation of the same function to reuse the runtime, and application-specific resources like code cache, libraries, and files. Thus, an invocation that arrives when a warm container is still in memory experiences a fast startup, with minimal initialization overhead. Note, however, that runtime and data *reuse* does not imply *sharing* — invocations that are active at the same time do not share the runtime or application data.

Photons are an extension of reusable warm containers. They allow safe sharing of the same container among multiple concurrent invocations of the same function. While current reusable containers can handle one invocation at a time, which causes unnecessary memory duplication, *photons* allow sharing runtimes, libraries, and application data across concurrent collocated invocations.

Checkpointing and process forking: One could fork a warm runtime to execute an invocation when it arrives. Inspired by Android's Zygote processes [17], this strategy has been explored in prior work [15, 23, 30, 34, 51], and can cut startup time to 2 ms as well as share the runtime memory using the copy-on-write mechanism. However, forking processes that are not designed with the fork operation in mind, like many runtimes and binaries, is also challenging from a correctness and consistency perspective. Thus, these proposals are restricted to a limited set of environments and runtimes, and as a consequence, reduce compatibility compared to general-purpose approaches [22]. Additionally, like warm containers, checkpointing and process forking approach does not address the sharing of application-specific data. Lastly, unless the serverless functions and runtimes are carefully customized, the copy-on-write mechanism used in this approach incurs significant memory overheads (§5.6).

Trading isolation for performance: One could obtain perfect resource and data sharing by executing multiple invocations within one container and providing runtime-level isolation. Similar ideas have been explored in prior work [27, 28, 45]. However, this requires either forgoing isolation between functions or severely restricting applications. For instance, CloudFlare, which uses this approach [7], relies on the Javascript V8 engine's isolation, and thus restricts applications to be purely Javascript, eschewing external libraries or binaries. This is obviously limiting – many such libraries are extremely popular for serverless use cases, *e.g., ffmpeg* for media encoding.

Our design philosophy: We observe that serverless workloads often feature concurrent executions of the same function by the same tenant. These invocations are *benign* towards each other. Today's warm containers take limited advantage of this trust. We thus present *Photons*, a framework for enabling resource sharing safely and without sacrificing generality in terms of runtimes and types of functions. Our approach involves separating private and shareable data for a function and carefully enforcing the separation of only the private data across invocations. This simple design does not preclude further enhancement with orthogonal techniques for reducing memory use and decreasing startup time, such as prioritizing the loading of critical parts of the environment on function initialization [38], pre-initializing and caching network end-points used by containers [49], and minimizing function memory footprint using Unikernels [44, 46, 47, 57].

4 *Photons*

In our framework, a *Photon* is a lightweight function executor that contains only the *private* state of a single invocation, while benefiting from sharing a runtime and common application state with other *photons*. For multiple invocations of the same function, *photons* effectively push the data isolation upwards through the stack, as shown in Figure 3, and the isolation across different functions is achieved using any of currently used serverless virtualization techniques [9, 13, 22, 48]. While this yields large performance, efficiency, and cost benefits (§5), it is non-trivial to achieve: we must ensure that in sharing the execution environment, *photons* do not interfere with either the correctness or performance of other *photons*' execution. In the following, we describe how a *photon*-enabled serverless platform works (§4.1), how we ensure correct execution of *photons* sharing an environment (§4.2), how shared state is identified (§4.3), and how we avoid performance interference (§4.4). Finally, we discuss what changes are required on the tenant-side and the provider-side to deploy *Photons* in the cloud (§4.5).

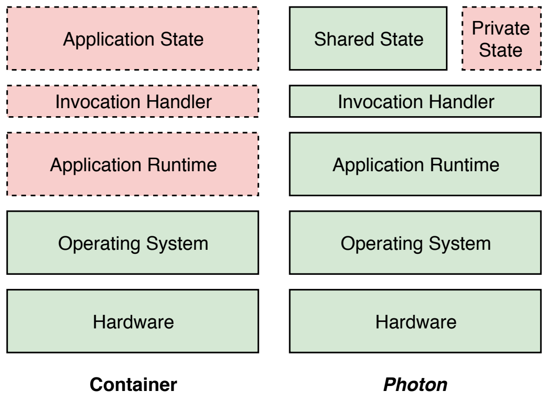


Figure 3: While today's serverless platforms use containers, virtualizing only up to the OS layer in the stack, Photons extend virtualization into the application runtime.

4.1 A *Photon*-enabled Serverless Platform

Photons are a generic primitive implementable on any serverless platform and for arbitrary language runtimes. However, for clarity, we describe them here in the context of our JVM implementation running atop a modified version of OpenWhisk [2]. OpenWhisk is a popular opensource serverless platform, with components that control, e.g., the submission of function invocations, the creation of containers to host invocations, or the storing of results. It also provides automatic scaling features to accommodate increasing function invocation load. Functions are registered by uploading the function source code (for Java functions, a compiled Jar archive) and are executed in language-specific container images that contain an Invocation Handler (see Figure 3), the component responsible for receiving requests to register and execute functions.

Photons are designed to have a small footprint and startup fast. To that end, we re-utilize already existing threads from previous *photons* to execute by keeping an active thread pool. During the execution, a *photon* can spawn multiple threads, as well as subprocesses, similar to today's serverless platforms. Further, all *photons* within the same execution environment share the same object heap, saving the overhead of allocating private memory for each *photon*, and making it possible for *photons* to easily share application state. Finally, all *photons* also share the application runtime code cache, meaning that all the optimized code produced during the code warmup phase (including code interpretation, profiling, and compilation to native code) benefits all *photons*, resulting in faster execution.

4.2 Data isolation

Photons maintain the semantics of existing serverless runtimes: invocations are isolated, even if they share the runtime. To provide data separation among multiple function executions within the same runtime, a write to a static field of a program must be performed on a local, invocation-specific copy of the field. Further, static initializers must be run independently for every new function execution, and finally, upon the termination of function execution, its local field copies must be deleted to avoid memory leaks.

Function loader: To support data separation, preserve the current serverless abstraction, and allow users a smooth transition to *Photons*, we implement a function loader that intercepts and instruments the user bytecode. The function loader automatically inserts appropriate operations and modifies access to global static program elements. The bytecode transformation is performed at class loading time by installing a class loader based on Javassist [32]. This class loader will load all application classes and ensure that all static fields are properly isolated. Note that our approach does not modify the runtime – the function loader is a just wrapper around JVM and is responsible for: (1) bytecode transformation; (2) initialize static elements; and (3) cleanup private state.

1. Code transformation: Private state (local copies of static fields) is built by replacing all static fields in the application code by static tables that return a local field copy given a unique function execution identifier (the generation of this identifier is discussed in §4.3). In addition to replacing all static field declarations, all reads and writes are also modified to use the local version of the field that is accessible through the table. Final static fields are ignored as no further modifications are possible, making it impossible to violate data separation through these.

Algorithm 1 is a simplified version of the transformations used to isolate private state. First, static tables are introduced, one for each static field (line 15). Then, for the class static initializer (line 16), constructors (line 18), and methods (line 20), all static field accesses are forwarded through the static table (line 7 and 9). We omit the full implementations of the two `Convert_Map_*` functions, but these convert a field write, *o.f = v*, into *o.f.put(photon_id, v)*, and a field read, *v = o.f*, into *v = o.f.get(photon_id)*, respectively. Finally, static fields are removed (line 22).

2. Running private initializers: A class static initializer is composed of a sequence of instructions that is guaranteed by the JVM to be executed before an instance of the respective class is created, a static field is accessed, or a static method is invoked. By design, a static initializer is only run once for the entire lifetime of the application execution. Thus, to ensure data separation across multiple function invocations within the same runtime, static initializers need to be executed independently for each. This is necessary to ensure that all local versions of static fields have been declared in the respective tables and have been properly initialized.

Algorithm 1 Data Separation at Class Loading Time

```
1:  $static\_initializers \leftarrow []$
2:  procedure ISOLATE_FIELD_ACCESSES($code\_block$)
3:    for $field\_access$ in $code\_block$ do
4:      $field \leftarrow field\_access.field$
5:      if isStatic($field$) and notFinal($field$) then
6:        if isWrite($field\_access$) then
7:          $Convert\_Map\_Write(field\_access)$
8:        else
9:          $Convert\_Map\_Read(field\_access)$
10: procedure LOAD_CLASS($class$)
11:   $fields\_to\_remove \leftarrow []$
12:   for $field$ in $class$ do
13:     if isStatic($field$) and notFinal($field$) then
14:       $fields\_to\_remove.add(field)$
15:       $class.addField(Map.class, field.name())$
16:   $Isolate\_Field\_Accesses(class.static\_initializer)$
17:   for $constructor$ in $class$ do
18:     $Isolate\_Field\_Accesses(constructor)$
19:   for $method$ in $class$ do
20:     $Isolate\_Field\_Accesses(method)$
21:   for $field$ in $fields\_to\_remove$ do
22:     $class.remove(field))$
23:   $initializer \leftarrow Clone(class.static\_initializer)$
24:   for $field\_access$ in $initializer$ do
25:     $field \leftarrow field\_access.field$
26:     if isFinal($field$) and isWrite($field\_access$) then
27:       $initializer.remove(field\_access)$
28:   $static\_initializers.add(initializer)$
```

To this end, we clone the class static initializer into a method that can be invoked every time a new *photon* is created (line 23). Also, we remove the initialization of static *final* fields from the cloned static initializer as these should only be initialized once across function invocations (line 27).

3. Cleaning up private state: When sharing the application runtime, we must clean up each function execution's private state to avoid memory leaks (*i.e.,* keeping reachable but unused memory alive). In particular, all local versions of static fields need to be removed from the respective tables in order to clean the state of an already finished function execution. To achieve that automatically, we make use of weak tables. A weak table is a common programming language abstraction that keeps a value reachable as long as there is a strong reference to the corresponding key. By relying on weak tables to store local static fields, we rely on garbage collection to automatically clean up unreachable local fields whenever a function execution context becomes unreachable (which occurs upon the termination of the function execution).

Limitations: Class loaders and bytecode are usually subject to order and assumptions, which might be invalidated if the application itself installs yet another class loader, or uses reflection. In these situations, users must modify the code manually and change access to static fields. From our experience, such changes are needed only infrequently and entail only small, localized code edits. In our test workloads, manual changes were required for only one library, where fewer than 10 lines of code across two classes in the MinIO library needed to change, as they used reflection to access static fields.

4.3 Sharing the Application State

Together with data isolation, it is also important to allow collocated invocations to efficiently share state. Since the shareable state will be visible (for both reads and writes) to multiple *photons* at the same time, we provide a set of abstractions to help developers manage the shared state. Besides allowing developers to share application state, these primitives can also be used to coordinate access to shared *resources*, like the file system. We keep these abstractions deliberately simple, such that they are sufficient to enable *photons* to share application state; more advanced and developer-friendly abstractions may, of course, be built atop these.

Shared object store: This is a key-value map for shared objects. This map resides in the same address space as all concurrent function invocations and can be used to share data with a small performance overhead.

Exclusive access: This is a locking primitive that can be used to coordinate accesses to the shared object store. For example, when a specific object needs to be inserted into the shared object store, a lock on the store can be acquired to ensure no data races. In addition, it is also possible, using the locking primitive, to easily build fine-grained read/write locks to further reduce contention when accessing specific objects in the shared object store.

Unique identifier: This unique *photon* identifier can be used to create a private temporary state. For example, if the function needs to use a temporary file in the local file system, the *photon* identifier could be used to identify a private file or folder.

Listing 1 presents a simple example of a Java *photon* function class. The `main` method contains three arguments: (a) a JSON map with function arguments, (b) an object store used to share state with other *photons*, and (c) an identifier. Line 14 shows how the *photon* identifier can be used to create a private temporary file and lines 16 to 23 show an example of how a shared resource can be initialized or fetched from the shared object store. For simplicity, we make use of `synchronized` blocks available in Java but more complex locking schemes could also be used (fine-grained locking, for example). Other languages also provide similar locking primitives.

```java
class MyServerlessFunction {
private static String run(
  MyModel model,
  String input,
  File pvtfile) { ... }

public static JsonObject main(
  JsonObject args,
  Map<String, Object> store,
  String photonId)
 {
  MyModel model;
  String input = args.getString("input");
  File pvtfile = new File("/tmp/" + photonId);

  synchronized (store) {
    if (!store.containsKey("model")) {
      model = new MyModel(...);
      store.put("model", model);
    } else {
      model = store.get("mode");
    }
  }

  String result = run(model, input, pvtfile);
  JsonObject response = new JsonObject();
  response.addProperty("predicted", result);
  return response;
}}
```

Listing 1: Example of a Photon function.

External isolation: *Photons* provide data isolation only within the runtime, while collocated invocations can still share the same file system without restrictions. This approach eases the data sharing across invocations, but at the same time creates a compatibility issue with today's platforms. For instance, if a function changes a global Linux environment variable, that creates a race condition across invocations and causes correctness issues.

To avoid external conflicts and inconsistencies, users can leverage exclusive access primitives and unique identifiers provided by *Photons* within the runtime to coordinate changes to the external system elements.

4.4 Vertical and horizontal scaling

Since *photons* remove the strict performance isolation across collocated invocations that is present on major serverless platforms, it is necessary to address potential resource bottlenecks and performance interference.

Vertical scaling: Using *photons*, we concentrate more function invocations in the same application runtime. While n invocations do not require n times the resources, as would be the case with separate containers, *some* scale-up is necessary. To scale automatically, *Photons* require two input parameters:

- *Initial container size*: Amount of CPU and memory required for every fresh container that hosts the first invocation with the entire runtime.
- *Container increment*: Additional CPU and memory required for every future collocated invocation.

As we show experimentally in §5.4, the maximum number of collocated invocations depends on the resources required per invocation and the current resource availability on a particular physical machine. Our experimental results align with the straightforward analysis in §2.3.

It is also important to allow users to specify requirements for CPU and memory separately because many workloads require little private state (memory) with significant demand for CPU cycles, *e.g.*, media conversion, or machine learning inference.

Increasing the resource provisioning of a running container is widely supported by mainstream container technology, but these resource updates must also be propagated to the application runtime. Such vertical scalability of application runtimes is still an active research topic, but recent work has shown how JVM memory can be scaled up and down in such contexts [29][5]. Some runtimes, such as JavaScript's, already support vertical memory scaling.

Horizontal scaling: *Photons* support a scale-out mechanism similar to today's serverless functions. For an incoming invocation, if a warm container exists, but the physical machine underneath does not have sufficient resources for a new *photon*, the invocation will be scheduled on another machine where a new container will be created.

Limitations: If a particular invocation requires more CPU or memory than initially estimated, in today's serverless platforms, that would degrade only its own performance. However, while using *Photons*, such an outlier can have a negative impact on other collocated invocations. This emphasizes the importance of correct resource requirement estimation and puts more responsibility on the users. However, note that cloud users already resolve similar provisioning challenges in systems like traditional Web servers, databases, and other

[5]This proposal has already been partially integrated into JDK and is now a feature in the mainstream Oracle JVM 9, and can be enabled by simply using a flag [12]. A fully patched JVM can be found here: https://github.com/jelastic/openjdk

applications that process multiple requests within the same memory space and without explicit performance isolation.

4.5 *Photon* deployment in the cloud

Deploying *Photons* in the cloud requires various changes both at the tenant as well as provider side.

Tenant side: Soft memory isolation provided by *Photons* does not suit all serverless workloads, *e.g.*, when sensitive data is processed in each invocation or a function executes untrusted code. We thus depend on the cloud tenants to mark their functions as Photons-friendly, the default being otherwise. Besides providing a Photon-friendly flag, tenants have to benchmark their workload to estimate CPU and memory requirements and provide them to the cloud provider, similar to today's serverless platforms. Also, when deploying a *Photon* function to the cloud, it is the user's responsibility to *compile* the function using the function loader we provide, to isolate global program elements as described in §4.2.

Provider side: Cloud providers have to modify their serverless scheduling policy to support *Photons*. They have to track active containers for every invocation that is currently running and collocate new invocations within active containers if the physical machine underneath has enough resources for CPU and memory increments. Cloud providers already implement a similar policy for tracking warm containers, so extending existing schedulers with *Photon* should be a minimal overhead.

4.6 Putting it all together

We summarize the end-to-end integration of a *photon* into serverless platforms as follows:

- The developer uses the abstractions provided (§4.3) to specify and manage state that is shared across *photons*.
- *Photons* automatically ensure data separation (§4.2).
- The developer profiles the application and identifies its resource requirements for the first invocation, and for successive invocations in the same execution context. We show how this can be done experimentally (§5.4).
- The service provider uses containers that haven't reached the concurrency limit to run additional invocations of the same function.
- The service provider vertically scales up container and runtime resources as more concurrent invocations execute within a shared container/runtime.

4.7 Other Application Runtimes

Besides Java, other languages such as Python and JavaScript are also very popular for developing serverless applications. We thus discuss how *photons* could be implemented for languages other than Java.

Implementing support for *photons* requires only two elements: (a) data separation, transparent to developers; and (b) abstractions to allow developers to share application state. JavaScript is single-threaded, and therefore, data separation is provided by design. Consider, for instance, the V8 runtime, one of the most popular JavaScript engines, developed by Google. In V8, JavaScript threads run inside an Isolate, a separate memory region owned by a specific thread. Photons are conceptually similar to V8 isolates with several differences: Photons share the garbage collector and code cache, and also allow for sharing program components between collocated invocations.

To support *Photons* atop V8, one would need to extend V8 with a shared data store to share operational data across invocations. Further, loading each invocation in a separate Isolate eliminates the need for code transformation because data separation is provided by the Isolates themselves. However, Isolates and separated memory heaps complicate library sharing. Thus, either we trade library sharing for no code transformation, or we apply similar techniques as for JVM and host multiple invocations within the same isolate using code transformation.

Python does not support memory isolates and is similar to Java, *i.e.,* data separation must be enforced so that the access to global variables and fields is restricted to the respective *photon*. Creating the shared object storage is also very similar to what we proposed.

Thus, there are no major roadblocks to implementing *photons* for other application runtimes. *Photons* rely on simple abstractions that can be easily found in most application runtimes. Besides, the design and implementation of *photons* did not require any modification to the underlying application runtime (JVM in this case) further demonstrating that *photons* can be implemented as a framework portable across different runtime implementations.

5 Evaluation

We evaluate several aspects of *photons* across five test workloads (§5.1) on a small testbed. In addition, we use large-scale simulations with production traces to evaluate the cluster-wide impact of *photons*. Our evaluation addresses the following questions:

§5.2: By how much do *photons* cut memory consumption compared to today's serverless platforms? By testing *photons* using production traces, we show 30% reduction in cluster-wide memory utilization.

§5.3: What benefits do *photons* offer in terms of reducing the number of cold starts? *Photons* achieve 52-75% reduction in the number of cold starts. This is particularly useful for latency-critical workloads that observe 3-100× improvement in average completion time when using *photons*.

§5.4: How do *photons* scale and what is the impact on throughput, response time, and cost? *Photons* save memory *without* performance degradation. Further, if desired, *Photons* also allow sacrificing the latency by 5% to decrease the overall cost by 35%.

§5.5: What overheads does the instrumentation required for data separation incur? On a micro-benchmark executing only one function per container, *photons* cause under 5% reduction in average throughput compared to unmodified OpenWhisk. This small overhead is more than compensated by the efficiencies of concurrent executions — all the above results already account for this.

§5.6: What is the overhead of systems that implement memory sharing by leveraging the copy-on-write mechanism? Compared to these approaches, we show that even for relatively simple functions, *photons* require 7× less memory.

5.1 Experimental setup

We evaluated a mix of five diverse workloads:

Sleep(1s): We use a function that sleeps for one second as the base case, one which largely measures the overheads of the infrastructure where the functions are invoked.

REST API (HTTP request): We implement representative API calls from a microservice benchmark [36]. These calls usually have a very short execution time and require minimal resources. This function simply passes a query to a backend database, after a small data transformation.

Hash(file): We fetch a 2 MB file from MinIO storage and hash its content. This function represents data processing pipelines that leverage massive parallelism by partitioning large data sets into multiple smaller chunks and invoking one function per chunk [26, 42].

Classify(image): We run an image classification workload given the current active research [31, 37, 40, 52] and industry [4, 21] focus on analyzing and running machine learning inference on serverless. The function loads an Inception v3 model (50 MB) from MinIO storage using the TensorFlow library. The model is pre-trained for ImageNet classification. After the model is loaded, each function loads an image ($\tilde{4}$ KB) from storage and performs classification. Note that the ML model can be shared across function executions because inference does not modify it.

Transform(video): Recent work proposed using serverless functions for implementing video transformations [25]. The idea is to split a video file into multiple small chunks and process each chunk in parallel. We mimic this workload by dividing a large video file into 10 second chunks. Each function execution fetches a single chunk from MinIO storage. We use the *ffmpeg* external binary to reduce the resolution of the chunk from 1280x720 to 640x480. The *ffmpeg* library is 63 MB in size and can be shared across function executions. However, this is a CPU intensive workload that creates a large private state of 120 MB by processing an input video file of 3.4 MB, and produces an output file of 1.8 MB.

We evaluate two aspects of Photons: (1) the performance of individual Photons; and (2) effects that Photons have at the cluster scale.

Machine-local evaluation: Using a single large machine, we show that Photons reduce the memory footprint for various workloads and scale-up without performance degradation. These effects are machine-local and more machines will not change the results as the functions are stateless, and machines do not communicate. We deployed the five above workloads on Apache OpenWhisk 0.9 with *Photons* being executed atop JVM 8. Our setup has one OpenWhisk node equipped with Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz with 2 sockets (8 cores each) and 252 GB of RAM. Another node with the same configuration mimics cloud block storage by running MinIO 6.0.8. The two machines communicate over a 10 Gbps dedicated link. Each of them runs Debian 9, with all containers being Docker 19.03.2 instances.

Cluster-wide evaluation: Since cluster-wide behavior depends on the distribution of job types and sizes, their arrival rate, etc., we use production traces published by Microsoft Azure from their serverless platform [54] and use them in a large scale cluster simulator we developed for this purpose.

The overall cluster utilization oscillates around 60%. Each invocation lasts between 1 ms and 10 min with memory requirements between 100 MB and 1800 MB. The distribution of function invocation frequency is skewed, with 1% of the functions being responsible for 90% of all invocations. Unfortunately, since the Microsoft study publishes only aggregated data without information on individual functions, there are two important parameters missing for fully realistic simulation. First, we assume the cold start time for each function follows a Normal distribution with a mean of 300 ms and a variance of 60 ms, based on a recent study [55]. Second, we need the amount of shareable memory across invocations. Since the smallest serverless functions are often simple API calls that implement simple logic atop a shareable runtime and the size of the smallest functions in Azure traces is around 100 MB, we assume that the shareable memory for each function is also normally distributed (mean = 100 MB, variance = 20 MB). This assumption can be seen as pessimistic given the amount of shareable memory in our test workloads.

We simulate a cluster of 100 machines with 40 GB of memory for 15 min. Using the parameters drawn from the Microsoft study, we generate 10,000 *unique* functions, and execute a total of 1.8 million function invocations.

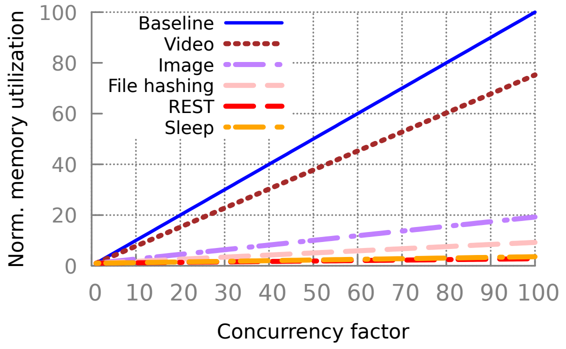


Figure 4: Total memory consumption of 5 types of functions for multiple concurrent requests on our testbed. The numbers are normalized to the case with one invocation per container with no data sharing (baseline). The baseline performance corresponds to OpenWhisk default performance as well as today's serverless platforms.

5.2 Memory consumption

Sharing runtime and app-state across functions translates to a reduction in per-function memory use. As discussed in §2.3, this reduction is bounded by: (a) the maximum number of concurrent functions executed per single container; and (b) the fraction of shareable vs. private memory in a function.

Figure 4 shows memory utilization with and without *photons*. All numbers are normalized to their respective baselines, *i.e.,* OpenWhisk, without *photons* for each workload. The "Baseline" increases one-to-one with concurrent function executions, irrespective of the workload because the normalization is with respect to each workload separately. In the absence of shared contexts with *photons*, each execution uses its own container. This is the behavior observed in all major serverless platforms [18].

The limiting factor that determines the memory footprint reduction is the concurrency factor — how many invocations are scheduled within the same container. As we show in §5.4, the runtime itself is not a bottleneck and can scale up to hundreds of concurrent invocations. However, the memory reduction is heavily affected by the function invocation arrival and the probability of two or more invocations of the same functions being created concurrently.

For evaluating the benefits of using *Photons* on the cluster memory utilization, we simulate cluster behavior using the previously described Microsoft Azure traces. While the overall memory utilization oscillates around 60% using default serverless platforms, *Photons* reduce the utilization by 30%[6], as shown in Figure 5.

This reduction does not directly reduce the used machines by 30% because *Photons* does not reduce CPU consumption

[6]We count the memory occupied by warm containers as free since it can be reclaimed at any time for other functions.

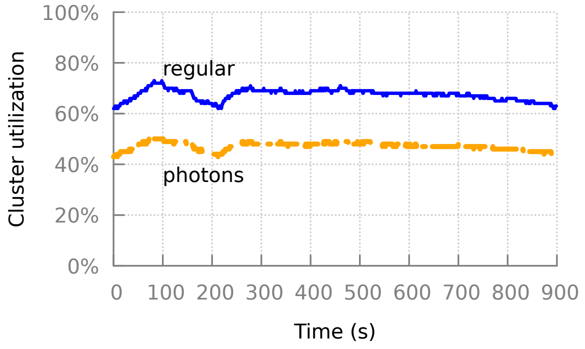


Figure 5: Total cluster memory utilization with and without photon-enabled sharing of execution contexts.

significantly. However, the memory savings allow keeping more warm containers and reduce the total number of slow starts, as we show next.

5.3 Cold starts

Besides memory utilization benefits, *photons* also reduce cold starts in two ways: (a) if an invocation is scheduled in an already active container, the cold start is skipped; and (b) lower memory utilization means that more containers can be kept warm in memory.

The reduction in cold starts depends on the invocation distribution. We thus evaluate two models: (a) the invocations are independent of each other, *i.e.,* an invocation of a function F does not increase the probability of another invocation of F arriving (*Poisson* arrival of invocations of each function); and (b) invocations of the same function are correlated in a Markovian manner, as indicated in the Microsoft traces for some functions [54]. The Markov model features more concurrent invocations of each function, and thus obviously allows greater opportunity for concurrent execution within a shared context.

We couple these models with the same simulation parameters as in §5.2. We find that *photons* reduce the total number of cold start events across all invocations by 52% compared to regular serverless platforms[7] for the Poisson model, and by 75% for the Markov model.

Avoiding cold start is essential for latency-critical workloads (< 100 ms). Thus, we measure execution time improvement (speedup) for a particular function invocation as the ratio of its execution time for a regular deployment and in a *photon* deployment. For every function, we then compute the average speedup across all of its invocations and show the results in Fig. 6. The execution time improvement depends on a function invocation's execution time. For long

[7]Regular serverless platforms reuse containers when possible, thus avoiding some cold starts. However, they do not support concurrent invocations within the same container, and incur cold starts for invocations that arrive when no warm container is available.

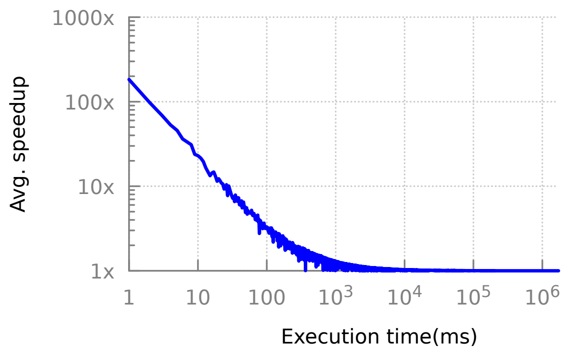


Figure 6: Photons reduce the total number of cold starts, even with independent invocation arrival (Poisson), as shown here. This leads to significant speedup, especially for short invocations. Although large invocations also avoid the cold start using photons, relative improvement is small. The Markov model results are strictly better.

invocations, the startup time is less material, while for short invocations, it is a substantial fraction of the total execution time. Thus avoiding cold start leads to more than 3× speedup on average for workloads shorter than 100 ms. For workloads shorter than 10 ms, we see an average speedup of more than 20×. Given that short invocations are likely to be more latency sensitive, this large speedup is of substantial value to such invocations. Furthermore, no invocation observes performance degradation by using *photons*. In the worst case, *photons* match the performance of today's serverless platforms (minus the < 5% overhead described in § 5.5).

The Azure traces used in our experiments are collected on a busy cluster, with a wide range of workloads. If the cluster utilization reduces, or the workload is suddenly skewed toward large functions with little shareable memory, the benefit of using *Photons* will shrink, but the performance will never be worse compared to today's platforms. However, if the workload consists of a large number of tiny functions or functions with large shareable memory, *Photons* should provide substantially higher benefits in terms of memory utilization and the number of cold start events compared to results based the Azure workload that we report here.

5.4 Scaling

To prevent performance from deteriorating as we collocate more invocations within the same runtime, we have to understand the resource requirements of each invocation and scale the CPU and memory allocation accordingly.

The first step in this process is to determine the resources needed for one isolated invocation. Note that this step is needed on today's platforms as well. Memory requirements are relatively simple to determine by running the workload without memory restrictions and detecting the peak memory

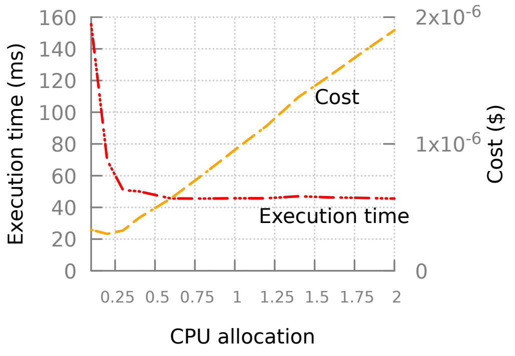


Figure 7: Container CPU allocation should be a result of the performance cost tradeoff. For the file hashing workload shown here, a configuration that provides minimal cost with decent performance is 0.2 CPU cores.

utilization throughout the execution. On the other hand, CPU requirements are more challenging to determine, since they usually present a trade-off between workload execution time and the cost — small CPU allocation leads to long execution time, but potentially, in some cases, lower total cost.

Fig. 7 illustrates our exploration of this trade-off for the *file hashing* workload. As we increase the CPU allocation, the execution time reduces as well as the cost[8]. However, this positive cost trend changes around 0.2 CPU. This happens due to the fact that a large portion of the *file hashing* workload is I/O operations (fetch/store a file) and more CPU just increases the cost with marginal execution time benefits. Thus, we pick 0.2 CPU as an operational point for this workload. By measuring the peak memory utilization, we see that one isolated invocation requires 127 MB in total for the runtime, libraries, and the private state.

Next, we have to determine resource requirements for every following invocation that will be collocated. To estimate the memory increment (private state), atop a warm runtime, we start adding concurrent invocations one by one and measure the increase in memory utilization. After several invocations, we use the peak increase as the private memory size. In the case of *file hashing*, we run 100 invocations and detect the private state as 3.5 MB.

Scaling the CPU allocation for collocated invocations is also simple. For avoiding any increase in execution time, we can just allocate additional c_{init} CPU cycles per additional invocation, where c_{init} is the amount of CPU allocated for the initial container. The intuition behind this approach is that the amount of work per invocation does not change, regardless of the number of invocations collocated. In the case of *file hashing*, we increase CPU by 0.2 CPU cores per invocation and memory by 3.5 MB.

[8]In this experiment we use the same pricing as on AWS Lambda [5].

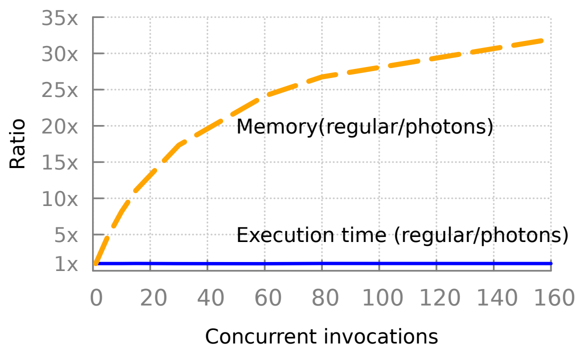


Figure 8: With 32 cores, our testbed can host 160 (160×0.2 *CPU cores) concurrent file hashing invocations. As we increase the number of invocations, the performance (execution time) of photons remains the same as on today's platforms with isolated containers. However, the memory saving grows with more invocations collocated.*

Workload	***CPU***	***Mem. initial***	***Mem. increment***
Sleep	0.1	49 MB	1 MB
REST	0.1	56 MB	1.5 MB
Hash	0.2	127 MB	3.5 MB
Classify	0.7	248 MB	29 MB
Video	1	160 MB	120 MB

Table 1: CPU and memory resources required for each of the workloads to scale without increased execution time.

The proposed scaling approach provides the same performance as running each invocation is a separate container with dedicated resources. This is illustrated in Fig. 8. We gradually increase the number of concurrent invocations of the same function (in this case *file hashing*), and measure average throughput and latency of OpenWhisk in two settings — regular and *photons*, and report the ratio between these two configurations. The ratio of 1 means that the execution time does not change with more concurrent requests, but the memory utilization obviously does because *photons* use significantly less memory.

We observe similar behavior across other workloads we tested. The results are summarized in Table 1. No workload observes performance degradation due to CPU contention as the number of collocated invocation increases, similarly to the experiment shown in Fig. 8.

Performance vs. cost: So far we analyzed *photons* that match the performance of today's platforms while using less memory. It is also possible to trade performance in terms of slightly longer execution time for a significantly lower cost by not scaling CPU proportionally to the number of collocated invocations. Namely, instead of adding c_{init} CPU cycles per new invocation, we could assign less. This is especially useful for workloads where CPU cycles are spent primarily on warming up the runtime, with the rest of the function executing I/O operations.

To illustrate this, we demonstrate how our target workloads scale with $0.5 \times c_{init}$ CPU increments and show results in Fig. 9. Although all workloads observe a reduction in cost, as expected, CPU intensive tasks like image classification observe significant performance degradation. On the other hand, I/O dependant tasks like REST continue scaling with a marginal overhead. Note that workloads that utilize the entire CPU like *video* cannot benefit from such optimization because a lack of CPU cycles would reflect proportionally in the execution time, making the total cost unchanged.

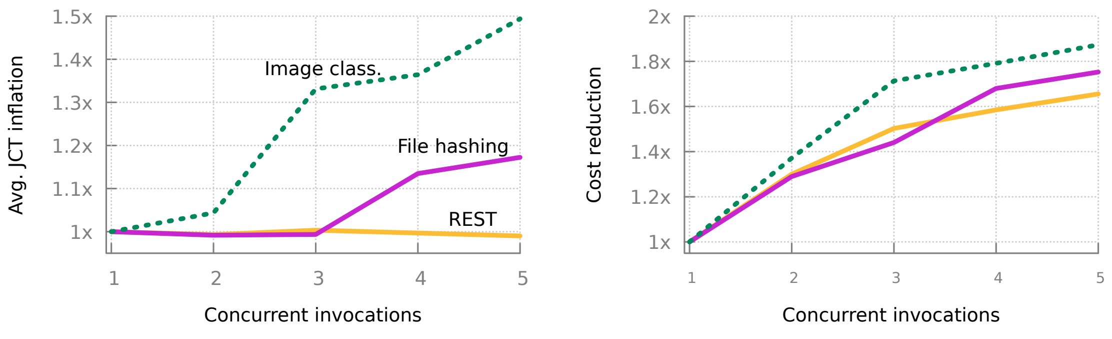
Although this cost-performance tradeoff is typical for systems in general and not specific to *photons*, it is important to expose the tradeoff to users, especially for tiny workloads hosted on heavy runtimes, like for instance REST on JVM. Namely, REST requires relatively high CPU allocation (10%) for handling the first invocation and spawning the runtime during the cold start phase. However, further invocations mostly do I/O intensive operations and require much less CPU time. This way *photons* decouple runtime overhead from the useful work and allow allocating resources needed for the runtime only once per container, both in terms of memory and CPU.

5.5 Data separation overhead

Virtualizing the runtime comes at the cost of enforcing data separation. As described in §4.2, data separation is provided by creating execution-local fields and by modifying all static field accesses to access the local copy instead. Local field copies are stored in a static table indexed by the *photon* identifier. This section shows the overhead introduced by accessing static fields through a static table instead of accessing them directly.

Each of our workloads except *Sleep* uses between 32 (*Hash*) to 38 (*Video*) static fields with 70 to 99 accesses to these fields respectively, showing that data separation is, in fact, necessary to provide the execution environment with the correct application semantics. For *Hash*, the static fields come from the MinIO communication library; for *Classify*, from the Java TensorFlow implementation; and for *Video*, from the *ffmpeg* wrapper for Java [8].

Data isolation incurs a small throughput overhead. *Hash* incurs a slightly higher overhead (4.8%) compared to *Classify* (1.7%), *REST* (1%) and *Video* (2.5%). This is due to *Classify* and *Transform* spending a large percentage of the execution in using the native libraries, which are unaffected by static field accesses. All our results already account for these overheads.



(a) *Execution time* (b) *Cost*

Figure 9: Instead of scaling allocated CPU for c_{init} per invocation, we assign c_{init} and every further invocation receives $0.5 \times c_{init}$. In I/O intensive workloads like REST and file hashing, for marginal performance degradation in terms of job execution time (left), we can achieve a significant reduction in cost (right). However, for CPU intensive workloads like image classification, the cost-benefit does not justify the performance loss.

Lastly, we note that in terms of achieving data separation automatically (instead of needing manual changes), across our workloads, we found only two classes which required manual intervention: in these two cases, Java reflection was being used to access a static field, something our implementation doesn't yet handle automatically.

5.6 Copy-on-write overhead

Copy-on-write is a powerful operating system mechanism for sharing memory while forking processes. A forked process will utilize the same memory pages as the parent process for read operations. However, for every write operation, a copy of a particular memory page has to be made and assigned to the forked process exclusively. This mechanism has been used in the context of serverless computing to share memory efficiently [23, 30, 34, 51].

We test the difference in memory consumption between copy-on-write systems and *photons* on the file hashing workload running with JVM. First, we warm up a container with 1000 consecutive invocations to give enough time to the runtime to populate code cache and other operational data structures. This creates a *zygote* runtime. Next, we fork the zygote runtime to handle one invocation and measure the total memory that has been changed in the process. A file hashing invocation pollutes 33 MB which is for 7× more than what is needed with *photons*. Although copy-on-write presents a significant improvement over today's platforms (3×), the overhead is still significant compared to *photons*.

6 Conclusion

To take advantage of massive cloud parallelism, many serverless computing workloads involve large numbers of concurrent invocations of the same function code. We observe that in many cases, these invocations replicate large amounts of state, including the language runtime, libraries, and shared state like machine learning models.

Photons exploit this redundancy by sharing the execution context across multiple invocations of the same function. While superficially simple, achieving this requires ensuring execution correctness and preventing performance inference across invocations. Addressing these challenges, *photons* reduce memory consumption (by 25-98% across our workloads). *Photons* also reduce the number of cold starts (by 52%), as well as the overall cluster memory use (by 30%) compared to today's platforms.

While we implemented *photons* for JVM, the ideas extend easily to other runtimes and languages. Our work also sets up several opportunities for future research: (a) co-executing *different* functions from the same tenant could unleash greater benefits, but would also require work on scheduling a mix of functions in a runtime, and resource sharing across them; (b) allowing the use of the shared object store as a cache across phases of serverless data processing pipelines; and (c) how to leverage the CPU-memory imbalance in serverless workloads for cloud scheduling.

Acknowledgments

We thank our shepherd Trevor Carlson and the reviewers for their feedback. Vojislav Dukic was supported, in part, by Google and the ETH Zürich Foundation. Rodrigo Bruno was supported in part by a grant from Oracle Labs.

References

[1] 2009 (Accessed May 25, 2020). *Understanding Container Reuse in AWS Lambda.* https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/.

[2] (Accessed May 25, 2020). *Apache OpenWhisk.* http://openwhisk.apache.org/.

[3] (Accessed May 25, 2020). *Apply machine learning models in Azure Functions with Python and TensorFlow.* https://docs.microsoft.com/en-us/azure/azure-functions/functions-machine-learning-tensorflow.

[4] (Accessed May 25, 2020). *Architecture of a Serverless Machine Learning Model.* https://cloud.google.com/solutions/architecture-of-a-serverless-ml-model.

[5] (Accessed May 25, 2020). *AWS Lambda pricing.* https://aws.amazon.com/lambda/pricing/.

[6] (Accessed May 25, 2020). *AWS Lambda web tutorial.* https://aws.amazon.com/getting-started/hands-on/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/.

[7] (Accessed May 25, 2020). *Cloud Computing without Containers.* https://blog.cloudflare.com/cloud-computing-without-containers/.

[8] (Accessed May 25, 2020). *FFMPEG for Java.* https://github.com/bramp/ffmpeg-cli-wrapper.

[9] (Accessed May 25, 2020). *gVisor.* https://gvisor.dev/.

[10] (Accessed May 25, 2020). *How to Deploy Deep Learning Models with AWS Lambda and Tensorflow.* https://aws.amazon.com/blogs/machine-learning/how-to-deploy-deep-learning-models-with-aws-lambda-and-tensorflow/.

[11] (Accessed May 25, 2020). *How to serve deep learning models using TensorFlow 2.0 with Cloud Functions.* https://cloud.google.com/blog/products/ai-machine-learning/how-to-serve-deep-learning-models-using-tensorflow-2-0-with-cloud-functions.

[12] (Accessed May 25, 2020). *JEP 346: Promptly Return Unused Committed Memory from G1.* http://openjdk.java.net/jeps/346.

[13] (Accessed May 25, 2020). *Kata Containers.* https://katacontainers.io/.

[14] (Accessed May 25, 2020). *Knative serverless framework.* https://knative.dev/.

[15] (Accessed May 25, 2020). *KNIX Microfunctions.* https://github.com/knix-microfunctions/knix/.

[16] (Accessed May 25, 2020). *OpenFaaS serverless framework.* https://www.openfaas.com/.

[17] (Accessed May 25, 2020). *Overview of memory management.* https://developer.android.com/topic/performance/memory-overview.

[18] (Accessed May 25, 2020). *Security Overview of AWSLambda.* https://d1.awsstatic.com/whitepapers/Overview-AWS-Lambda-Security.pdf.

[19] (Accessed May 25, 2020). *Serverless Networking.* https://github.com/serverlessunicorn/ServerlessNetworkingClients.

[20] (Accessed May 25, 2020). *Understanding serverless cold start.* https://azure.microsoft.com/en-us/blog/understanding-serverless-cold-start/.

[21] (Accessed May 25, 2020). *Use AWS DeepLens to give Amazon Alexa the power to detect objects via Alexa skills.* https://aws.amazon.com/blogs/machine-learning/use-aws-deeplens-to-give-amazon-alexa-the-power-to-detect-objects-via-alexa-skills/.

[22] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *USENIX NSDI.*

[23] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-performance Serverless Computing. In *USENIX ATC.*

[24] Gene M. Amdahl. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS.*

[25] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. 2018. Sprocket: A serverless video processing framework. In *ACM SoCC.*

[26] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. *Serverless Computing: Current Trends and Open Problems.*

[27] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. 2018. Putting the" Micro" back in microservice. In *USENIX ATC.*

[28] Stefan Brenner and Rüdiger Kapitza. 2019. Trust more, serverless. In *SYSTOR.*

[29] Rodrigo Bruno, Paulo Ferreira, Ruslan Synytsky, Tetiana Fydorenchyk, Jia Rao, Hang Huang, and Song Wu. 2018. Dynamic Vertical Memory Scalability for OpenJDK Cloud Applications. In *ISMM.*

[30] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *EuroSys.*

[31] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2018. A Case for Serverless Machine Learning. In *NeurIPS.*

[32] Shigeru Chiba and Muga Nishizawa. 2003. An easy-to-use toolkit for efficient Java bytecode translators. In *International Conference on Generative Programming and Component Engineering.*

[33] Eyal de Lara, Carolina S Gomes, Steve Langridge, S Hossein Mortazavi, and Meysam Roodi. 2016. Hierarchical serverless computing for the mobile edge. In *IEEE/ACM Symposium on Edge Computing (SEC).*

[34] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *ASPLOS.*

[35] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-latency Video Processing Using Thousands of Tiny Threads. In *USENIX NSDI.*

[36] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS.*

[37] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das. 2019. Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud. In *IEEE CLOUD.*

[38] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *FAST.*

[39] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651* (2018).

[40] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2018. Serving deep learning models in a serverless platform. In *IEEE International Conference on Cloud Engineering (IC2E).*

[41] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *ACM SoCC.*

[42] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* (2019). arXiv:1902.03383

[43] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *USENIX OSDI*.

[44] Ricardo Koller and Dan Williams. 2017. Will Serverless End the Dominance of Linux in the Cloud?. In *HotOS*.

[45] James Larisch, James Mickens, and Eddie Kohler. 2018. Alto: lightweight vms using virtualization-aware managed runtimes. In *MPLR*.

[46] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library operating systems for the cloud. In *ACM SIGARCH Computer Architecture News*.

[47] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *SOSP*.

[48] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. In *Linux journal*.

[49] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. In *HotCloud*.

[50] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *ACM SIGMOD*.

[51] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-optimized Containers. In *USENIX ATC*.

[52] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. 2018. Serverless computing for container-based architectures. In *Future Generation Computer Systems*.

[53] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2019. Starling: A Scalable Query Engine on Cloud Function Services. *arXiv preprint arXiv:1911.11727* (2019).

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

[55] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *USENIX ATC*.

[56] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. *USENIX ATC*.

[57] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: a dynamic library operating system for simplified and efficient cloud virtualization. In *USENIX ATC*.