From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers

Sadjad Fouladi Francisco Romero Dan Iter Qian Li
Shuvo Chatterjee+ Christos Kozyrakis Matei Zaharia Keith Winstein

Stanford University, +Unaffiliated

Abstract

We present gg, a framework and a set of command-line tools that helps people execute everyday applications—e.g., software compilation, unit tests, video encoding, or object recognition—using thousands of parallel threads on a cloud-functions service to achieve near-interactive completion times. In the future, instead of running these tasks on a laptop, or keeping a warm cluster running in the cloud, users might push a button that spawns 10,000 parallel cloud functions to execute a large job in a few seconds from start. gg is designed to make this practical and easy.

With gg, applications express a job as a composition of lightweight OS containers that are individually transient (lifetimes of 1–60 seconds) and functional (each container is hermetically sealed and deterministic). gg takes care of instantiating these containers on cloud functions, loading dependencies, minimizing data movement, moving data between containers, and dealing with failure and stragglers.

We ported several latency-sensitive applications to run on gg and evaluated its performance. In the best case, a distributed compiler built on gg outperformed a conventional tool (`icecc`) by 2–5×, without requiring a warm cluster running continuously. In the worst case, gg was within 20% of the hand-tuned performance of an existing tool for video encoding (ExCamera).

1 Introduction

Public cloud-computing services have steadily rented out their resources at finer and finer granularity. Sun's Grid utility (2005), Amazon's EC2 (2006), and Microsoft's Azure virtual machines (2012) began by renting virtual CPUs for a minimum interval of one hour, with boot-up times measured in minutes. Today, the major services will rent a virtual machine for a minimum of one minute and can typically provision and boot it within 45 seconds of a request.

Meanwhile, a new category of cloud-computing resources offers even finer granularity and lower latency: cloud functions, also called serverless computing. Amazon's Lambda service will rent a Linux container to run arbitrary x86-64 executables for a minimum of 100 milliseconds, with a startup time of less than a second and no charge when it is idle. Google, Microsoft, Alibaba, and IBM have similar offerings.

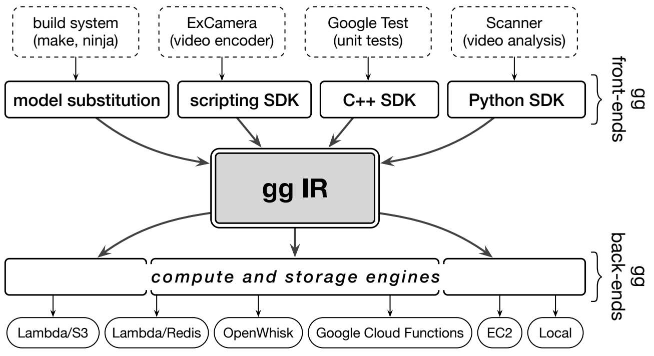


Figure 1: gg helps applications express their jobs as a composition of interdependent Linux containers, and provides back-end engines to execute the job on different cloud-computing platforms.

Cloud functions were intended for asynchronously invoked microservices, but their granularity and scale has allowed researchers to explore a different use: as a burstable supercomputer-on-demand. These new systems launch a *burst-parallel* swarm of thousands of cloud functions, all working on the same job. The goal is to provide results to an interactive user—much faster than can be accomplished on the user's own computer or by booting a cold cluster, and cheaper than maintaining a warm cluster for occasional tasks.

Recent work has validated this vision. ExCamera [15] and Sprocket [3] launch thousands of cloud functions, with inter-thread communication over TCP, to encode, search, and transform video files quickly. PyWren [23] exposes a Python API and uses AWS Lambda functions for linear algebra and machine learning. Serverless MapReduce [35] and Spark-on-Lambda [36] demonstrate a similar approach.

Unfortunately, building applications on swarms of cloud functions is difficult. Each application must overcome a number of challenges endemic to this environment: (1) workers are stateless and may need to download large amounts of code and data on startup, (2) workers have limited runtime before they are killed, (3) on-worker storage is limited, but much faster than off-worker storage, (4) the number of available cloud workers depends on the provider's overall load and can't be known precisely upfront, (5) worker failures occur when running at large scale, (6) libraries and dependencies differ in a cloud function compared with a local machine, and (7) latency to the cloud makes roundtrips costly. Past applications have addressed only subsets of these challenges, in application-specific ways.

In this paper, we present gg, a general framework for building burst-parallel cloud-functions applications, by building them on an abstraction of transient, functional containers, or *thunks*. gg helps applications express their jobs in terms of interrelated thunks (hermetically sealed, short-lived containers that may reference the output of other thunks or produce other thunks as output), then schedules, instantiates, and executes those thunks on a cloud-functions service.

gg can containerize and execute existing programs, e.g., software compilation, unit tests, video encoding, or searching a movie with an object-recognition kernel. gg does this with thousands-way parallelism on short-lived cloud functions. In some cases, this yields considerable benefits in terms of performance. Depending on the frequency of the task (e.g., for compilation or unit tests every few minutes), cloud functions are also much less expensive than keeping a comparable cluster running continuously.

gg and other parallel execution systems. In its goals and approach, gg is kin with container-orchestration systems such as Kubernetes [5] and Docker Swarm [10], outsourcing tools like the Utility Coprocessor [12] and `icecc` [20], and cluster-computation tools such as Hadoop [38], Dryad [22], Spark [40], and CIEL [27].

But gg also differs from these systems in its focus on a new computing substrate (cloud functions), mode of execution (burst-parallel, latency-sensitive programs starting from zero), and target application domain (everyday "local" programs, e.g. software compilation, that depend on an environment captured from the user's own laptop).

For example, the "stateless" nature of cloud functions (they boot up with no dependable transient state) makes gg place a large focus on efficient containerization and dependency management: loading the minimal set of the right files into each container at boot-up. Cluster-computation systems like Dryad, Spark, and CIEL do not do this—although they can interface with existing code and systems (e.g., a video encoder or a database server), these components must be loaded in advance by the user on a long-lived compute node. Container systems like Kubernetes do this, but they are not aimed at efficient execution of a transient interactive task—gg is more than 45× faster than Google Kubernetes Engine at startup, and 13× faster than Spark-on-Lambda (Figure 7). We discuss related work more completely in Section 2.

1.1 Summary of Results

We ported four applications to express their jobs in gg's format: a description of each container, and how it depends on other containers, that we call the *intermediate representation*,

Compiling Inkscape

Tool	Time	Cost
single-core `make`	32m 34s	—
`icecc` to a warm 48-core EC2 machine	6m 51s	\$2.30/hr
`icecc` to a warm 384-core EC2 cluster	6m 57s	\$18.40/hr
gg to AWS Lambda	1m 27s	50¢/run

Figure 2: Compiling Inkscape using gg on AWS Lambda is almost 5× faster than outsourcing the job to a warm 384-core cluster, without the costs of maintaining a warm cluster for an occasional task.

or IR (§3). One of them does it automatically, by inferring the IR from an existing software build system (e.g., `make` or `ninja`). The rest write out the description explicitly: a unit-testing framework (Google Test [17]), parallel video encoding with inter-thread communication (ExCamera [15]), and object recognition using Scanner [30] and TensorFlow [1].

We then implemented gg back-ends, which interpret the IR and execute the job, for five compute engines (a local machine, a cluster of warm VMs, AWS Lambda, IBM Cloud Functions, and Google Cloud Functions) and three storage engines (S3, Google Cloud Storage, and Redis) (Figure 1).

For compiling large programs from a cold start, gg's functional approach and fine-grained dependency management yield significant performance benefits. Figure 2 shows a summary of the results for compiling an open-source software, Inkscape [21]. Running "cold" on AWS Lambda (with no pre-provisioned compute resources), gg was almost 5× faster than an existing system (`icecc`), running on a 48-core or 384-core cluster of warm VMs (i.e., not including time to provision and boot the VMs[1]).

In summary, gg is a practical tool that addresses the principal challenges faced by burst-parallel cloud-functions applications. It helps developers and users build applications that burst from zero to thousands of parallel threads to achieve low latency for everyday tasks. gg is open-source software and the source code is available at https://snr.stanford.edu/gg.

2 Related Work

gg has many antecedents—cluster-computation systems such as Hadoop [38], Spark [40], Dryad [22], and CIEL [27]; container orchestrators like Docker Swarm and Kubernetes; outsourcing tools like `distcc` [8], `icecc` [20], and UCop [12]; rule-based workflow systems like `make` [13], CMake [7], and Bazel [4]; and cloud-functions tools like ExCamera/mu [15], PyWren [23], and Spark-on-Lambda [36].

Compared with these, gg differs principally in its focus on targeting a new computing substrate (thousands of cloud functions, working to accelerate a latency-sensitive local-

[1]Current cloud-computing services typically take an additional 0.5–2 minutes to provision and boot such a cluster.

application task). We discuss how gg fits with the prior literature in several categories:

Process migration and outsourcing. The idea of accelerating a local application's interactive operations by using the resources of the cloud has a long pedigree; earlier work such as the Utility Coprocessor (UCop) also sought to "improve performance from the coffee-break timescale of minutes to the 15–20 second timescale of interactive performance" by outsourcing to a cloud VM [12]. gg shares the same goal.

gg's architectural differences from this work come from its different needs: instead of outsourcing applications transparently to a *single* warm cloud VM, gg orchestrates *thousands* of unreliable and stateless cloud functions from a cold start. Unlike UCop, gg is not transparent to the application—we require applications to be ported to express jobs in gg's format. In return, gg provides optimized orchestration of swarms of cloud functions and fault tolerance (failed functions are rerun with the same inputs). Unlike UCop's distributed caching filesystem, gg's IR, which is based on content-addressed immutable data, allows cloud workers to be provisioned with all necessary dependencies in a single roundtrip and to communicate intermediate values directly between each other.

Container orchestration. gg's IR resembles container and environment-description languages, including Docker [10] and Vagrant [34], and container-orchestration systems such as Docker Swarm and Kubernetes. In contrast to these systems, gg's thunks are designed to be efficiently instantiated within a cloud function, expressible in terms of other thunks to form a computation graph, and deterministic and defined by their code and data, allowing gg to provide fault tolerance and memoization. These systems were not designed for transient computations, and gg has much quicker startup. For example, starting 1,000 empty containers with gg takes about 4 seconds on a VM cluster or on AWS Lambda. Google Kubernetes Engine, given a warm cluster, takes more than 3 minutes (§5.1). Recent academic work has shown how to lower this overhead to provide faster cloud-functions services [28].

Workflow systems. Workflow systems like Dryad [22], Spark [40], and CIEL [27] let users execute a (possibly dynamic) DAG of tasks on a cluster. However, gg differs from these systems in some significant ways:

- gg is aimed at a different kind of application. For example, while Spark is often used for data analytics tasks, it is not commonly used for accelerating the sorts of "everyday" local applications that gg is designed for. No prior work has successfully accelerated something like "compiling Chromium" using Spark, and the challenges in accomplishing this (capturing the user's local environment and the information flow of the task, exporting the job and its dependencies efficiently to the cloud, running thousands of copies of the C++ compiler in a fault-tolerant way) are simply not what Spark does.
- gg uses OS abstractions: it encapsulates arbitrary code and dependency files in lightweight containers, somewhat similar to a tool like Docker. gg focuses on efficiently loading code and its minimal necessary dependencies on cloud functions that boot up with no dependable state. By contrast, systems like Dryad and Spark principally use language-level mechanisms. While their jobs can interface with existing software (e.g., the Dryad paper [22] describes how a node can talk to a local SQL Server process, and Spark jobs routinely invoke system binaries such as `ffmpeg`), these systems do not take care of deploying the existing code, worrying about how to move the container in a way that minimizes bytes moved across the network, etc. The user is responsible for loading the necessary code and dependencies beforehand on a pool of long-lived machines.
- gg is considerably lighter weight. In practice, attempts to port workflow systems to support execution on cloud functions (scaling from zero) have not performed well, partly because of these systems' overheads. Because of its focus on transient execution, gg carries an order-of-magnitude less overhead. For example, gg is 13× faster at invoking 1,000 "sleep 2" tasks than Spark-on-Lambda (Figure 7).
- gg supports dynamic data access (a function can produce another function that accesses arbitrary data) and non-DAG dataflows (e.g., loops and recursion). It does this while remaining agnostic to the application's programming language. For example, gg has no language-level API binding to launch a new subtask. (CIEL also allows subtasks to spawn new subtasks, but requires use of its Skywriting programming language to do this.)

Burst-parallel cloud functions. Researchers and practitioners have taken advantage of cloud-functions platforms to implement low-latency, massively parallel applications. ExCamera [15] uses AWS Lambda to scale out video encoding and processing tasks over thousands of function invocations, and PyWren [23] exposes a MapReduce-like Python API that executes on AWS Lambda. Spark-on-Lambda [40] is a port of Spark that uses AWS Lambda cloud functions. In contrast, gg helps applications use cloud-functions platforms for a broader set of workloads, including irregular execution graphs and ones that change as execution evolves. gg's main contribution is specifying an IR that permits a diverse class of applications (written in any programming language) to be abstracted from the compute and storage platform, and to leverage common services for dependency management, straggler mitigation, and scheduling.

Build tools. Several build systems (e.g., `make` [13], Bazel [4], Nix [11], and Vesta [19]) and outsourcing tools (such as `distcc` [8], `icecc` [20], and `mrcc` [26]) seek to incrementalize, parallelize, or distribute compilation to more-powerful remote machines. Building on such systems, gg automatically transforms existing build processes into their own IR. The goal is to compile programs quickly—irrespective of the software's own build system—by making use of cloud-functions platforms that can burst from complete dormancy to thousands-way parallelism and back.

Existing remote compilation systems, including `distcc` and `icecc`, send data between a master node and the workers frequently during the build. These systems perform best on a local network, and add substantial latency when building on more remote servers in the cloud. In contrast, gg uploads all the build input once and executes and exchanges data purely within the cloud, reducing the effects of network latency.

3 Design and Implementation

gg is designed as a general system to help application developers manage the challenges of creating burst-parallel cloud-functions applications. The expectation is that users will take computations that might normally run locally or on small clusters for a long time (e.g., test suites, machine learning, data exploration and analysis, software compilation, video encoding and processing), and outsource them to thousands of short-lived parallel threads in the cloud, in order to achieve near-interactive completion time.

In this section, we describe the design of gg's intermediate representation (§3.1), front-end code generators (§3.2), and back-end execution engines (§3.3).

3.1 gg's Intermediate Representation

The format that gg uses—a set of documents describing a container and its dependency on other containers—is intended to elicit enough information from applications about their jobs (fine-grained dependencies and dataflow) to be able to efficiently execute a job on constrained and stateless cloud functions. It includes:

1. A primitive of a content-addressed cloud *thunk*: a codelet or executable applied to named input data.
2. An intermediate representation (IR) that expresses jobs as a lazily evaluated lambda expression of interdependent thunks.
3. A strategy for representing dynamic computation graphs and data-access patterns in a language-agnostic and memoizable way, using tail recursion.

We discuss each of these elements.

3.1.1 Thunk: A Lightweight Container

In the functional-programming literature, a thunk is a parameterless closure (a function) that captures a snapshot of its arguments and environment for later evaluation. The process of evaluating the thunk—applying the function to its arguments and saving the result—is called *forcing* it [2].

For gg, our goal is to simplify the creation of new applications by allowing them to target the IR, which lets them leverage the common services provided by the back-end engines. Accordingly, the representation of a thunk follows from several design goals. It should be: (1) simple enough to be portable to different compute and storage platforms, (2) general enough to express a variety of plausible applications, (3) agnostic to the programming language used to implement the function, (4) efficient enough to capture fine-grained dependencies that can be materialized on stateless and space-limited cloud functions, and (5) able to be memoized to prevent redundant work.

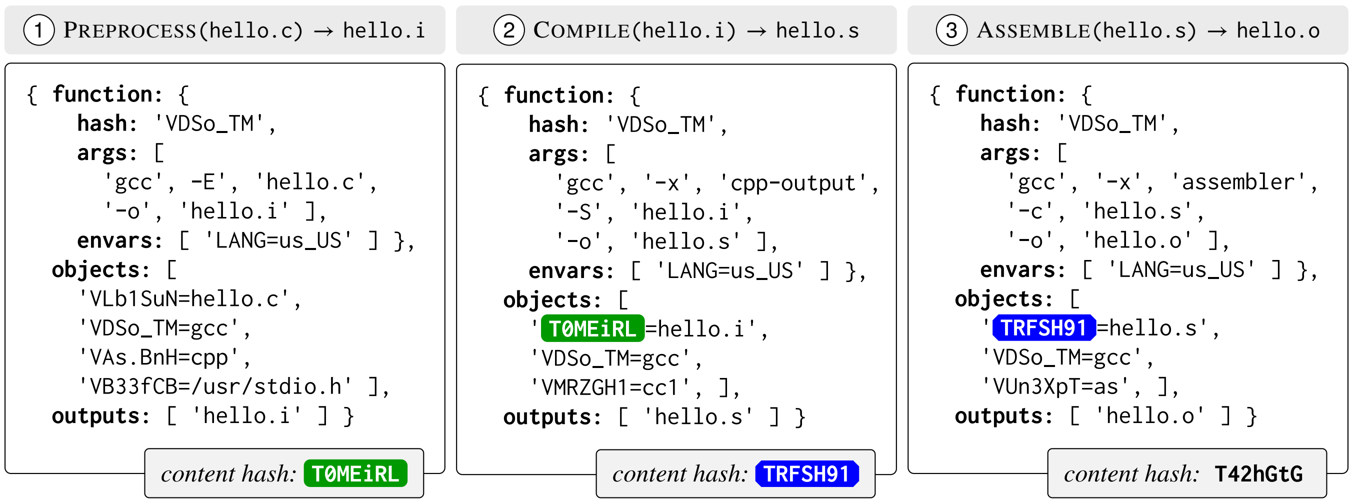
To satisfy these requirements, gg represents a thunk with a description of a container that identifies, in content-addressed manner, an x86-64 Linux executable and all of its input data objects. The container is hermetically sealed: it is not allowed to use the network or access unlisted objects or files. The thunk also describes the arguments and environment for the executable, and a list of tagged output files that it will generate—the results of forcing the thunk. The thunk is represented as a Protobuf [31] structure (Figure 3 shows three thunks for three different stages of a build process). This container-description format is simple to implement and reason about, and is well-matched to the statelessness and unreliability of cloud functions.

In the content-addressing scheme, the name of an object has four components: (1) whether the object is a primitive value (hash starting with `V`) or represents the result of forcing some other thunk (hash starting with `T`), (2) a SHA-256 hash, (3) the length in bytes, and (4) an optional tag that names an object or a thunk's output.

Forcing a thunk means instantiating the described container and running the code. To do this, the executor must fetch the code and data values. Because these are content-addressed, this can be from any mechanism capable of producing a blob that has the correct name—durable or ephemeral storage (e.g., S3, Redis, or Bigtable), a network transfer from another node, or by finding the object already available in RAM from a previous execution. The executor then runs the executable with the provided arguments and environment—for debugging or security purposes, preferably in a mode that prevents the executable from accessing the network or any data not listed as a dependency. The executor collects the output blobs, calculates their hashes, and records that the outputs can be substituted in place of any reference to the just-forced thunk.

3.1.2 gg IR: A Lazily Evaluated Lambda Expression

The structure of interdependent thunks is what defines the gg IR. We use a one-way IR, a document format that applications write to express their jobs, as opposed to a two-way API (e.g., a function call to spawn a new task and observe its result)



① PREPROCESS(hello.c) → hello.i

```
{ function: {
    hash: 'VDSo_TM',
    args: [
      'gcc', -E', 'hello.c',
      '-o', 'hello.i' ],
    envars: [ 'LANG=us_US' ] },
  objects: [
    'VLb1SuN=hello.c',
    'VDSo_TM=gcc',
    'VAs.BnH=cpp',
    'VB33fCB=/usr/stdio.h' ],
  outputs: [ 'hello.i' ] }
```

content hash: `T0MEiRL`

② COMPILE(hello.i) → hello.s

```
{ function: {
    hash: 'VDSo_TM',
    args: [
      'gcc', '-x', 'cpp-output',
      '-S', 'hello.i',
      '-o', 'hello.s' ],
    envars: [ 'LANG=us_US' ] },
  objects: [
    'T0MEiRL=hello.i',
    'VDSo_TM=gcc',
    'VMRZGH1=cc1', ],
  outputs: [ 'hello.s' ] }
```

content hash: `TRFSH91`

③ ASSEMBLE(hello.s) → hello.o

```
{ function: {
    hash: 'VDSo_TM',
    args: [
      'gcc', '-x', 'assembler',
      '-c', 'hello.s',
      '-o', 'hello.o' ],
    envars: [ 'LANG=us_US' ] },
  objects: [
    'TRFSH91=hello.s',
    'VDSo_TM=gcc',
    'VUn3XpT=as', ],
  outputs: [ 'hello.o' ] }
```

content hash: `T42hGtG`

Figure 3: An example of gg IR consisting of three thunks for building a "Hello, World!" program that represents the expression ASSEMBLE(COMPILE(PREPROCESS(hello.c))) → hello.o. To produce the final output hello.o, thunks must be forced in order from left to right. Other thunks, such as the link operation, can reference the last thunk's output using its hash, T42hGtG. Hashes have been shortened for display, and dependencies between thunks are shown in color.

because we expect the application will be running on the user's own computer, at some remote cloud-functions engine: the intention is to avoid roundtrips over a long-latency path by keeping the application out of the loop. We also envision that it will be possible to better schedule and optimize a job, and easier to maintain different interoperable back-ends, if the application is out of the loop before execution begins.[2] This representation exposes the computation graph to the back-end, along with the identities and sizes of objects that need to be communicated between thunks. Based on this information, the back-end can schedule the forcing of thunks, place thunks with similar data-dependencies or an output-input relationship on the same physical infrastructure, and manage the storage or transfer of intermediate results, without roundtrips back to the user's own computer.

The IR allows gg to schedule jobs efficiently, mitigate the effect of stragglers by invoking multiple concurrent thunks on the critical path, recover from failures by forcing a thunk a second time, and memoize thunks. This is achieved in an application-agnostic, language-agnostic manner.

The application generally starts by forcing a single thunk that represents the ultimate outcome of the interactive operation. This thunk typically depends on other thunks that need to be forced first, etc., leading the back-end to lazily force thunks recursively until obtaining the final result. Figure 3 shows an example IR for computing the expression ASSEMBLE(COMPILE(PREPROCESS(hello.c))).

3.1.3 Tail Recursion: Supporting Dynamic Execution

The above design is sufficient to describe a directed acyclic graph (DAG) of deterministic tasks executing in the cloud. However, many jobs do not have a data-access pattern that is completely known upfront. For example, in compiling software, it is unknown *a priori* which header files and libraries will need to be read by a given stage. Other applications use loops, recursion, and other non-DAG dataflows.

An application may also have an unpredictable degree of parallelism. For example, an application might detect objects in a large image, and then on each subregion where an object is detected (which may be zero regions, or might be 10,000 regions), the application searches for a target object. Here, the computation graph is not known in advance.

Systems like PyWren [23] and CIEL's Skywriting language [27] handle this case by giving tasks access to an API call to invoke a new task. For gg, we aimed to preserve the memoizability and language-independence of the IR, which is challenging if tasks can invoke tasks on their own and if gg must expose a language binding. Instead, gg handles this situation through language-independent tail recursion: a thunk can write another thunk as its output.

3.2 Front-ends

We developed four front-ends that emit gg IR: a C++ SDK, a Python SDK, a group of command-line tools, and a series

[2]Systems like the LLVM compiler suite [25] (which allows front-end language compilers to benefit from a library of back-end optimization passes and assemblers, interfacing through an IR) and Halide [33] (which separates an image-processing algorithm from its schedule and execution strategy) have demonstrated the benefits of a rigid representational abstraction in other settings. gg's use of an IR is not exactly the same as these, but it has a similar value in abstracting front-ends (applications and the tools that help them express their jobs) from back-end execution engines in a way that allows efficient and portable execution.

of *model substitution* primitives that can infer gg IR from a software build system.

The C++ and Python SDKs are straightforward. Each exposes a thunk abstraction and allows the developer to describe a parallel application in terms of codelets. These codelets are applied to blobs of named data, which may be read-only memory regions or files in the filesystem.

The model-substitution primitives extract a gg IR description of an existing build system, without actually compiling the software. Instead, we run the build system with a modified PATH so that each stage is replaced with a stub: a *model* program that understands the behavior of the underlying stage well enough so that when the model is invoked in place of the real stage, it can write out a thunk that captures the arguments and data that will be needed in the future, so that forcing the thunk will produce the exact output that would have been produced during actual execution. We used this technique to infer gg IR from the existing build systems for several large open-source applications (§4.1).

3.3 Back-ends

gg IR express the application against an abstract machine that requires two components: an *execution engine* for forcing the individual thunks, and a content-addressed *storage engine* for storing the named blobs referenced or produced by the thunks. The *coordinator* program brings these two components together.

Storage engine. A storage engine provides a simple interface to a content-address storage, consisted of GET and PUT functions to retrieve and store objects. We implemented several content-addressed storage engines, backed by S3, Redis, and Google Cloud Storage. We also have a preliminary implementation (not evaluated here) that allows cloud functions to communicate directly among one another, avoiding the latency and throughput limitations of using a reliable blob storage (e.g., S3) to exchange small objects.

Execution engine. In conjunction with a storage engine, each execution engine implements a simple abstraction: a function that receives a thunk as the input and returns the hashes of its output objects (which can be either values or thunks). The engine can execute the thunk *anywhere*, as long as it returns correct output hashes that are retrievable from the storage engine. We implemented back-end execution engines for several environments: a local multicore machine, a cluster of remote VMs, AWS Lambda, Google Cloud Functions, and IBM Cloud Functions (OpenWhisk).

The coordinator. The main entry-point for executing a thunk is the *coordinator* program. The inputs to this program are the target thunk, a list of available execution engines and the storage engine. This program implements services offered by gg, such as job scheduling, memoization, failure recovery and straggler mitigation.

Upon start, this program materializes the target thunk's dependency graph, which includes all the other thunks needed to get the output. Then, the thunks that are ready to execute are passed to execution engines, based on their available capacity. When the execution of a thunk is done, the program updates the graph by replacing the references to the just-forced thunk and adds a cache entry associating the output hash to the input hash. The thunks that become ready to execute are placed on a queue and passed to the execution engines when their capacity permits. The unified interface allows the user to mix-and-match different execution engines, as long as they share the same storage engine.

The details of invocation, execution and placement are left to the execution engines. For example, the default engine for AWS Lambda/S3 invokes a new Lambda for each thunk. The Lambda downloads all the dependencies from S3 and sets up the environment, executes the thunk, uploads the outputs back to S3 and shuts down. For applications with large input/output objects, the roundtrips to S3 could affect the performance. As an optimization for such cases, the user can decide to run the execution engine in the "long-lived" mode, where each Lambda worker stays up until the job finishes and seeks out new thunks to execute. The execution engine keeps an index of all the objects that are already present on each worker's local storage. When placing thunks on workers, it selects the worker with the most data available, in order to minimize the need to fetch dependencies from the storage back-end.

The coordinator can also apply optimizations to the dependency graph. For example, multiple thunks can be bundled as one and sent to the execution engine. This is useful when the output of one thunk will be consumed by the next thunk, creating a linear pipeline of work. By scheduling all of those thunks on one worker, the system reduces the number of roundtrips.

Failure recovery and straggler mitigation. In case of coordinator failure, the job can be picked up where it was left off, as the coordinator program uses on-disk cache entries to avoid redoing the work that has already been done. In case of a recoverable error in executing a thunk, the execution engine notifies the coordinator with the failure reason, where it can decide to retry the job or pass it to another available execution engine for execution.

Straggler mitigation is another service managed by the coordinator program which duplicates pending executions in the same or a different execution engine. The program keeps track of the execution time for each thunk, and if the execution time exceeds a timeout (set by either the user or the application developer) the job will be duplicated. Since the functions don't have any side-effects, the coordinator simply picks the output that becomes ready first.

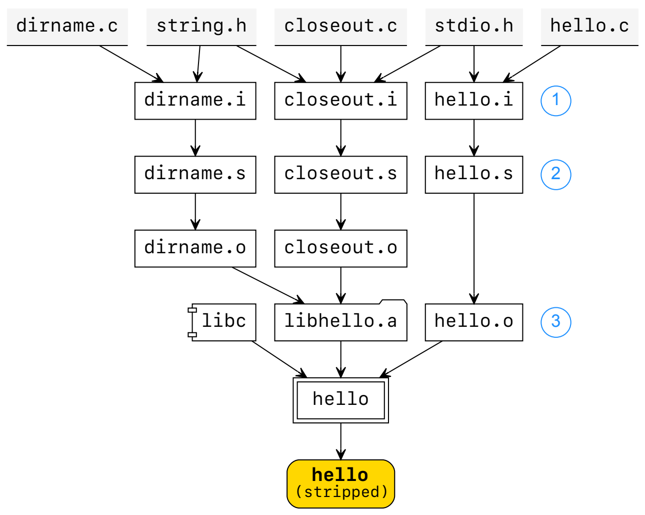


Figure 4: Part of the IR of interdependent thunks inferred with model substitution from the GNU `hello` build system. Each box represents a thunk and is labeled with the name of its output. The contents of the numbered thunks are depicted in Figure 3 (Many header files and other dependencies omitted for simplicity).

3.4 Implementation Notes

We implemented gg in about 14,000 lines of C++. The implementation consists of five compute engines (a local machine, a cluster of warm VMs, AWS Lambda, Google Cloud Functions, and IBM Cloud Functions), three storage engines (S3, Google Cloud Storage, and Redis), a series of command line tools to aid generation, execution and inspection of gg IR, a C++ and Python SDK, and several model programs for different stages of build process.

4 Applications

We used gg to implement several applications, each emitting jobs in the gg IR. We describe these in turn.

4.1 Software Compilation

The time required to compile software is an evergreen frustration for software developers; a popular cartoon even spoofs the duration of this job [39]. Today's open-source applications have grown larger and larger. For example, the Chromium Web browser takes more than four *hours* to compile on a four-core laptop computer from a cold start. Many solutions have been developed to leverage warm machines in a local cluster or cloud datacenter (e.g., `distcc` or `icecc`). We developed such an application on top of gg.

Using model substitution, we implemented models for seven popular stages of a C or C++ software build pipeline: the preprocessor, compiler, assembler, linker, archiver, indexer, and `strip`. These allow us to automatically transform some software build processes (e.g., a `Makefile` or `build.ninja` file) into an expression in gg IR, which can then be executed with thousands-way parallelism on cloud-functions platforms to obtain the same results as if the build system had been executed locally. Figure 4 illustrates the resulting IR from an example invocation (the enumerated thunks are detailed in Figure 3). These models are sufficient to capture the build process of some major open-source applications, including OpenSSH [29], Python interpreter [32], the Protobuf library [31], the FFmpeg video system [14], the GIMP image editor [16], the Inkscape vector graphics editor [21], and the Chromium browser [6].[3]

Build systems often include scripts that run in addition to these standard tools, such as a tool to generate configuration header files, but typically such scripts run upstream of the preprocessor, compiler, etc. Therefore, gg captures these script outputs by a model as dependencies.

Capturing dependencies of the preprocessor. Preprocessing is the most challenging stage to model. It requires not only capturing the source file as dependencies, but also all the header files that are both directly and indirectly included by that source file. Capturing *all* header files in a container is not feasible, because cloud functions are constrained in storage. For example, AWS Lambda has a 500 MB storage limit.

The precise header files required to preprocess a file can be discovered at fine grain, but only by invoking the preprocessor (i.e., `gcc -M`) which is an expensive operation at large scale. Finding the dependencies for each source file in Chromium takes nearly half an hour on a 4-core computer.

To solve this problem, the application uses gg's capabilities for dynamic dataflow at runtime. gg's preprocessor model generates thunks that do dependency inference in parallel on cloud functions. These thunks have access only to a stripped-down version of the user's include directories, preserving only lines with C preprocessor directives (such as `#include` and `#define`). These thunks then produce further thunks that preprocess a given source-code file by listing only the necessary header files.

4.2 Unit Testing

Software test suites are another application that can benefit from massive parallelism. Using gg's C++ SDK, we implemented a tool that can generate gg IR for unit tests written with Google Test [17], a popular C++ test framework used by projects like LLVM, OpenCV, Chromium, Protocol Buffers, and the VPX video codec library.

[3]We have to emphasize that no changes were made to the underlying build system of these programs. The main challenge here is to build correct and complete models for programs used in the build pipeline, such as `gcc` and `ld`, which is a one-time effort. However, an arbitrary build system may require other programs to be modeled, or execute these programs in an aberrant way that is outside of the scope of model substitution.

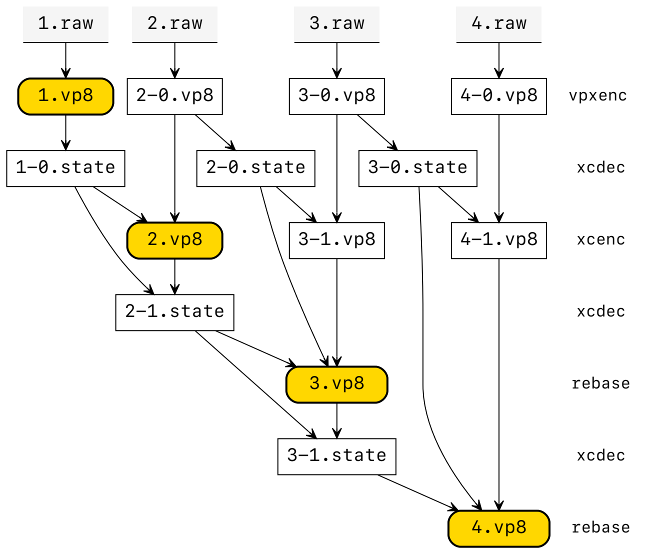


Figure 5: Visual representation of the gg IR for a video-processing workflow [15].

Typically, each test is a standalone program that can be run in parallel with other tests, with no dependency requirements between them. No changes to the code are necessary, with one exception: if a test case needs to access files on the file system, then the programmer has to annotate the test case with the list of files that it wants to access. This process can be automated by running the tests locally and then tracing the open system calls invoked by each test case. The tool uses these annotations, either handcrafted or automatically generated, to capture each test's dependencies. A separate thunk is created for each test case, allowing the execution engine to exploit the available parallelism.

4.3 Video Encoding

The ExCamera system [15] uses cloud-functions infrastructure to run interdependent video-processing tasks with 4,000-way parallelism [15]. Cloud workers exchange data through TCP connections, brokered by a tightly coupled back-end that was bound to AWS Lambda. To demonstrate gg's expressive power and performance, we ported ExCamera into a "front-end-only" version that targets gg IR.

In ExCamera, the functions necessary for parallel video encoding are ENCODE, DECODE, ENCODE-GIVEN-STATE, and REBASE. The algorithm first encodes each chunk in parallel using ENCODE and then, in a serial process, REBASEs each output on top of the state left by the previous chunk. Video-codec states must be communicated between workers in order to stitch together the overall video. Figure 5 shows the dependency graph for encoding a batch of four chunks.

The original ExCamera keeps Lambda workers warm by keeping the raw video in RAM and communicating video-codec states over TCP between workers. gg's back-end for AWS Lambda also keeps workers warm and keeps the raw video in their local filesystem. gg routes thunks to workers that already have the necessary data, but brokers inter-worker communication through S3. Finally, gg provides fault tolerance, which ExCamera's own back-end lacks.

4.4 Object Recognition

The increase in visual computing applications has motivated the design of frameworks such as Scanner [30], which is a system for productive and efficient video analysis at scale. To use Scanner, the user feeds in a compressed video and designates an operation to be applied on each decoded frame. To compare Scanner's execution engine with gg, we used the gg C++ SDK to implement a two-step analysis pipeline. In the first stage, the frames of a video V are decoded in batches of m frames, using DECODE(V,m) function. Subsequently, an object-recognition kernel, OBJECT-REC, is applied to the decoded frames and returns the top five recognized objects for each frame.

We implemented the DECODE function using FFmpeg [14] and implemented OBJECT-REC in TensorFlow's C++ API [1] using a pre-trained Inception-v3 model [37]. gg's thunks were able to bundle these pre-existing applications. We implemented the same pipeline in Scanner for comparison. To do so, we leverage Scanner's internal video decoder and the same TensorFlow kernel and pre-trained Inception-v3 model.

4.5 Recursive Fibonacci

To demonstrate the way that gg handles dynamic dataflows, we used the C++ SDK to implement a classic recursive Fibonacci program in the gg IR. The application is expressed using two functions: ADD(a,b), which returns the sum of its two input values and FIB(n) which recursively computes the n-th Fibonacci number as ADD(FIB($n-1$), FIB($n-2$)) or the base case when $n \leq 1$.

Figure 6 shows the execution steps. In the beginning, there is only one thunk, FIB(4). After execution, instead of returning a value, it returns three thunks, replacing the target with the sum of two preceding Fibonacci numbers. The IR expands (for the recursive case) and contracts (for the base case), until resolving to the final value.

In a naïve recursive implementation of the Fibonacci series, each Fibonacci value is evaluated many times. However, in gg, the functions are memoized and lazily-executed, resulting in each Fibonacci value computed only once.

5 Evaluation

We evaluated gg's performance by executing each application in gg, compared with comparable tailor-made or native applications. Although we implemented back-end engines for

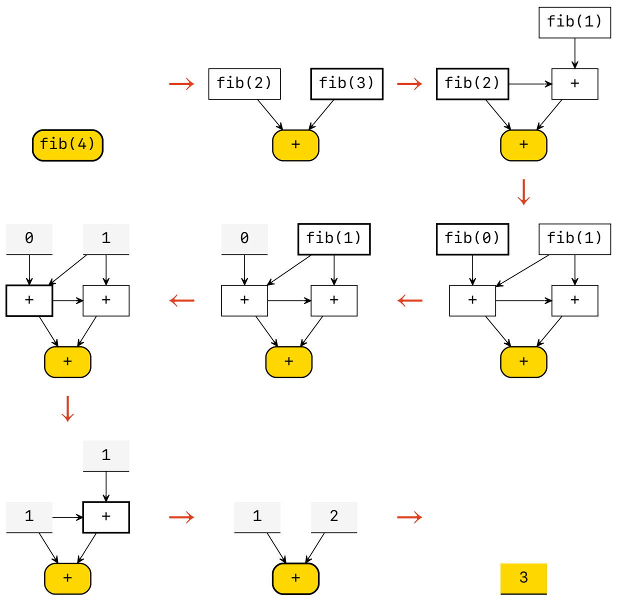


Figure 6: Evolution of the IR for a recursive Fibonacci application. Execution begins with a single thunk. As each thunk is forced, returning a new thunk or the base case, the IR expands and contracts. The engine lazily forces thunks until it can return the overall value.

1K trivial containers running "sleep 2"		
AWS Lambda	gg-λ	06s ± 01s
	PyWren	46s ± 08s
	Spark-on-Lambda	54s ± 21s
Google Kubernetes Engine	Kubernetes	03m 08s ± 03s

Figure 7: Comparison of completion time for running 1,000 `sleep(2)` tasks using four different systems. gg's lightweight design and implementation has less overhead than other systems.

Compiling Inkscape on AWS Lambda (total of 3602 thunks)	
Initial graph construction	56 ms
Mean time to read a thunk	188 µs ± 367 µs
Mean time to recompute the IR per thunk	336 µs ± 560 µs
Invocation (thunk completion to invocation of all dependent thunks)	142 ms ± 135 ms

Figure 8: gg's overheads allow for relatively fine-grained tasks.

several cloud-functions platforms (including Google Cloud Functions and IBM Cloud Functions), we found that AWS Lambda had the best performance and available parallelism. As a result, we focus on evaluation results from gg's AWS Lambda back-end.

5.1 Startup Overhead

To motivate the importance of gg's lightweight abstractions, we implemented a trivial job, 1,000 parallel tasks each executing `sleep(2)`, using four frameworks: gg, PyWren, Spark-on-Lambda, and Kubernetes. The first three frameworks were executed on AWS Lambda, and the last on Google Kubernetes Engine (GKE), which was given a warm cluster of eleven 96-core VMs (1,056 cores in total) on which to allocate containers. Figure 7 shows the results.

gg is able to quickly scale to 1,000 concurrent containers and finish the job 7.5–9× faster when compared with other two frameworks running on Lambda. After subtracting off the 2-second sleep time, this translates to 11–13× less overhead. For PyWren, on average, each worker spends 70% of its time on setting up the Python runtime (downloading and extracting a tarball). A large portion of this runtime consists of packages that are not used by our `sleep(2)` program (cf. gg fine-grained dependency tracking). Google Kubernetes Engine was not designed for transient computations and was not optimized for this use case; it is much slower to start 1,000 Docker containers.

Additionally, we measured the overheads associated with loading thunks and recomputing the IR after a thunk is done in Figure 8. These overheads, especially the invocation overhead, support an intuition about the appropriate granularity of thunks: gg works well when thunks last about 1–20 seconds each.

5.2 Software Compilation

To evaluate gg's application for software compilation, we measured the start-to-finish build times under multiple scenarios on a set of unmodified large open-source packages. We compared these times with existing tools under the same scenarios. For distributed builds outsourced from a 4-core EC2 VM, we found that gg is able to achieve significantly shorter build times than existing approaches.

5.2.1 Evaluation Set

To benchmark gg's performance, we picked four open-source programs written in C or C++: FFmpeg, GIMP, Inkscape, and Chromium. No changes were made to the code or the underlying build system of these packages. We compiled all packages with GCC 7.2.

All the 4-core machines used in the experiments are EC2 `m5.xlarge`, and all the 48-core machines are EC2 `m5.12xlarge` instances. To realistically simulate users sending applications to nearby datacenters, client machines reside in the US West (N. California) region, and outsource their jobs to machines in the US West (Oregon) region.

		Local (make)		Distributed (icecc)		**Distributed (gg)**	
	Estimated SLoC	1 core	48 cores	48 cores	384 cores	384 cores	AWS Lambda
FFmpeg	1,200,000	06m 19s	20s	01m 03s	39s	40s	44s ± 04s
GIMP	800,000	06m 48s	49s	02m 35s	02m 38s	01m 26s	01m 38s ± 03s
Inkscape	600,000	32m 34s	01m 40s	06m 51s	06m 57s	01m 20s	01m 27s ± 07s
Chromium	24,000,000	15h 58m 20s	38m 11s	46m 01s	42m 18s	40m 57s	18m 55s ± 10s

Figure 9: Comparison of cold-cache build times in different scenarios described in §5.2. gg on AWS Lambda is competitive with or faster than using conventional outsourcing (icecc), and in the case of the largest programs, 2–5× faster. This includes both the time required to generate gg IR from a given repository and then to execute the IR.

5.2.2 Baselines

For each package, we measured the start to finish build time in four different scenarios as the baseline for local and distributed builds:

`make`, `make` (48): The package's own build system was executed on a single core (`make`), and with up to 48-way parallelism (`make -j48`). The `make` and `make` (48) tests were done on 4-core and 48-core EC2 VMs, respectively. No remote machines were involved in these tests.

`icecc` (48), `icecc` (384): The package was built using the `icecc` distributed compiler on a 4-core client that outsources the job to a 48-core VM, or to eight 48-core VMs, for a total of 384 cores.

5.2.3 gg's Benchmarks

We conducted the following experiments for each package to evaluate gg:

1. **gg (384)**: The package was built with the same configuration as the `icecc` (384) experiment: a 4-core client farming out to eight 48-core machines, using gg's back-end for a cluster of VMs.
2. **gg-λ**: The package was built on a 4-core client outsourcing to AWS Lambda, using as many concurrent Lambdas as possible (up to 8,000 in the case of Chromium).

   For Chromium experiments, an additional standby EC2 VM acted as the *overflow* worker for thunks whose total data size exceeded Lambda's storage limit of 500 MB. Throughout building Chromium, there were only 2 thunks (out of ~90,000 thunks) that did not fit on a Lambda and had to be forced on this overflow node.

5.2.4 Discussion of Evaluation Results

Figure 9 shows the median times for the package builds. gg is about 2–5× faster than a conventional tool (`icecc`) in building medium- and large-sized software packages. For example, gg compiles Inkscape in 87 seconds on AWS Lambda, compared with 7 minutes when outsourced with `icecc` to a warm 384-core cluster. This is a 4.8× speedup. Chromium, one of the largest open-source projects available, compiles in under 20 minutes using gg on AWS Lambda, which is 2.2× faster than `icecc` (384).

We do not think gg's performance improvements on AWS Lambda can be explained simply by the availability of more cores than our 384-core cluster; `icecc` improved only modestly between the 48-core and 384-core case and doesn't appear to effectively use higher degrees of parallelism. This is largely because `icecc`, in order to simplify dependency tracking, runs the preprocessor locally, which becomes a major bottleneck. gg's fine-grained dependency tracking allows the system to efficiently outsource this step to the cloud and minimize the work done on the local machine.

Figure 10 shows an execution breakdown for compiling Inkscape. We observe two important characteristics. First, the large spikes correspond to Lambdas that have failed or taken longer than usual to complete. gg's straggler mitigation detects and relaunches these jobs to prevent an increase in end-to-end latency. Second, the last few jobs are primarily serial (archiving and linking), and consume almost a quarter of the total job-completion time. These characteristics were also observed in the other build jobs.

5.3 Unit Tests

To benchmark gg's performance in running unit tests created with the Google Test framework, we chose the VPX video codec library [9], which contains ~7,000 unit tests. We annotated each test with the list of required data files.

The Google Test library that is shipped with LibVPX is only capable of running the tests serially. To establish a better baseline, we used `gtest-parallel`, a program that executes Google Test binaries in parallel on the local machine. We ran the tests with 4- and 48-way parallelism and compared the results with gg on AWS Lambda, with 8,000-way parallelism. Figure 11 shows the summary of these results.

Using the massive parallelism available, gg was able to execute all of the test cases in parallel, and 99% of the test

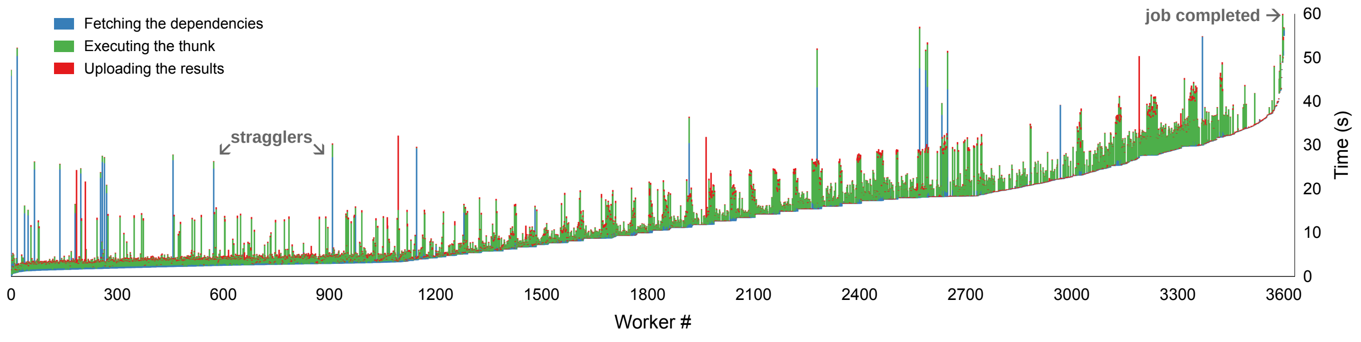


Figure 10: Breakdown of workers' execution time when building Inkscape using gg on AWS Lambda. Serial stages (archiving and linking) consume almost a quarter of the total job-completion time. Spikes indicate stragglers, which are mitigated by gg using standard techniques. In this experiment, stragglers mostly consist of Lambdas that have trouble communicating with the storage back-end (S3).

	gg-λ	gtest (4)	gtest (48)
LibVPX Test Suite	03m 25s	51m 45s	04m 40s

Figure 11: Running the LibVPX test suite using gg on AWS Lambda outperforms running the tests with 4-way and 48-way parallelism on a local machine. 99% of the test cases complete within 30 seconds.

	gg-λ	original
ExCamera	01m 30s	01m 16s

Figure 12: The gg version of ExCamera is 18% slower than the hand-optimized original ExCamera, which was written to pipeline I/O and computation within a Lambda worker.

cases finished within the first 30 seconds. From a developer's point of view, this improves turnaround time and translates into faster discovery of bugs and regressions.

5.4 Video Encoding

We evaluated the gg implementation of ExCamera on AWS Lambda, with the original implementation as the baseline. The selected configuration was the same as ExCamera's original paper (6 frames per chunk, 16 chunks per batch). The input video consisted of 888 chunks, and all chunks had been uploaded to S3 in raw format prior to execution. For the original ExCamera implementation, a 64-core VM (`m4.16xlarge`) was used as the rendezvous server to broker TCP streams between Lambda workers.

Figure 12 shows the results. The original ExCamera was hand-coded to pipeline I/O and computation to reduce end-to-end latency. By contrast, gg's abstract interface must force and load all data-dependencies before running user code, and cannot perform this optimization. ExCamera-on-gg is 18% slower than the original, but adds memoization and fault-tolerance, unlike the original ExCamera.

Object Recognition	
gg local (64 cores)	04m 30s
gg on AWS Lambda	37s
Scanner local (64 cores)	05m 39s
Scanner on cluster (140 cores)	03m 14s

Figure 13: Scanner-on-gg outperforms the original Scanner on the same hardware, and performs even faster on AWS Lambda.

5.5 Object Recognition

We compared the original Scanner [30] with the gg implementation using a 4K video with more than 6,000 frames. For the baseline, we chose the most favorable execution parameters through an exhaustive search. The optimal number of pipeline instances and frame batches were 14 and 75, respectively. Within each pipeline, each video chunk is first decoded into raw images before being passed to the TensorFlow kernel execution thread. Each execution thread only needs to load the model once per stream of frames. Scanner local was run on a 64-core machine (`m4.16xlarge`). Scanner on cluster was run with a 4-core master (`m4.xlarge`) and four 36-core workers (`c4.8xlarge`), of which Scanner uses 35 and leaves one for scheduling. For the gg implementation, the video was broken up into five-second chunks and uploaded to the cloud prior to execution. Each chunk was decoded in batches of 25 frames. For the object recognition task, the IR was configured to the optimal number of frame batches per task.

Figure 13 presents the summary of the results. While Scanner on cluster is 39% faster than gg local, it is 5.2× slower than gg on AWS Lambda. Scanner local is over 9× slower than gg on AWS Lambda. gg's lightweight scheduling and execution engine removes several layers of abstraction present in Scanner's design.

6 Limitations and Discussion

gg has a number of important limitations and opportunities for future work.

Direct communication between workers. Although commentators have noted that "*two Lambda functions can only communicate through an autoscaling intermediary service... like S3*" [18], our experience differs: we have found that on AWS Lambda, two Lambda functions can communicate directly using off-the-shelf NAT-traversal techniques, at speeds up to 600 Mbps (although the performance is variable and requires an appropriate protocol and failure-recovery strategy). We thus believe that the performance of systems such as ExCamera, PyWren, and gg is likely to improve in the future as practitioners develop better mechanisms for harnessing this computing substrate, including direct communication.

In follow-on work, we are developing a 3D ray-tracing engine on gg, that will quickly render complex scenes across thousands of nodes, where the scene geometry and textures consume far more space than any individual node's memory. To achieve sufficient performance, this will require low-latency and high-speed communication between workers, motivating the use of direct network connectivity, instead of an intermediate storage system such as S3 or Pocket [24].

Limited to CPU programs. gg specifies the format of the code as an x86-64 Linux ELF executable. The IR has no mechanism to signal a need for GPUs or other accelerators, and efficiently scheduling such resources poses nontrivial challenges, because loading and unloading configuration state from a GPU is a more expensive operation than memory-mapping a file. We plan to investigate the appropriate mechanisms for a gg back-end to schedule thunks onto GPUs.

A gg DSL to program for the IR. Currently, we have implemented a C++ and Python SDK for users to express applications that target the gg IR. However, this requires the user to explicitly provide an x86-64 executable and all of its dependencies prior to thunk generation. We envision a language in which users can write high-level code in Python or C++, using primitives such as a parallel map, fold, and other operations, which will be compiled into the gg IR.

Why cloud functions? Transient, burst-parallel execution on services like AWS Lambda produces a different cost structure from a warm cluster. It takes about the same amount of time for gg to compile Inkscape on AWS Lambda as on a 384-core cluster of warm EC2 VMs (Figure 9). The job costs about 50 cents *per run* on Lambda, compared with \$18.40 *per hour* to keep a 384-core cluster running (Figure 2). Whether it is financially beneficial for the gg user to run such jobs on long-running VMs or on cloud functions depends on how often the user has a job to run. From an economic perspective, the provider is compensating the infrequent user for their elasticity; e.g., for having structured their workload to vacate compute resources when no task is active, and to tolerate variations in the exact number of nodes available for a job and the timing of when they are allocated.

In the future, we expect the performance characteristics of VMs and Lambda-like services to move closer together. There is no intrinsic reason for it to take more than 30 seconds to provision and boot an infrastructure-as-a-service VM in the public cloud. Linux itself can boot in less than a second, and KVM and VMware can provision a VM in less than 3 seconds. We understand the remaining time is largely "management plane" overhead. If this can be reduced, then cloud functions may hold no compelling advantage over virtual machines for executing burst-parallel applications—but tools like gg that aid efficient execution on remote compute infrastructure (whether VM or cloud function) may remain valuable.

7 Conclusion

In this paper, we described gg, a framework that helps developers build and execute burst-parallel applications. gg presents a portable abstraction: an intermediate representation (IR) that captures the future execution of a job as a composition of lightweight Linux containers. This lets gg support new and existing applications in various languages that are abstracted from the compute and storage platform and from runtime features that address underlying challenges: dependency management, straggler mitigation, placement, and memoization.

As a computing substrate, we suspect cloud functions are in a similar position to Graphics Processing Units in the 2000s. At the time, GPUs were designed solely for 3D graphics, but the community gradually recognized that they had become programmable enough to execute some parallel algorithms unrelated to graphics. Over time, this "general-purpose GPU" (GPGPU) movement created systems-support technologies and became a major use of GPUs, especially for physical simulations and deep neural networks.

Cloud functions may tell a similar story. Although intended for asynchronous microservices, we believe that with sufficient effort by this community the same infrastructure is capable of broad and exciting new applications. Just as GPGPU computing did a decade ago, nontraditional "serverless" computing may have far-reaching effects.

Acknowledgments

We thank the USENIX ATC reviewers and our shepherd, Ed Nightingale, for their helpful comments and suggestions. We are grateful to Geoffrey Voelker, George Porter, Anirudh Sivaraman, Zakir Durumeric, Riad S. Wahby, Liz Izhikevich, and Deepti Raghavan for comments on versions of this paper. We also thank Alex Ozdemir for his measurements on Lambda networking, and Alex Poms for all his help with Scanner. This work was supported by NSF grant CNS-1528197, DARPA grant HR0011-15-2-0047, and by Google, Huawei, VMware, Dropbox, Facebook, and the Stanford Platform Lab.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from http://tensorflow.org.

[2] Harold Abelson and Julie Sussman, G. J. with Sussman. *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, Cambridge, 2nd editon edition, 1996.

[3] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *ACM Symposium on Cloud Computing (SoCC 2018)*, Carlsbad, CA, 2018.

[4] Bazel build system. https://bazel.build.

[5] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Queue*, 14(1):10:70–10:93, January 2016.

[6] The Chromium browser. https://www.chromium.org/Home.

[7] CMake. https://cmake.org.

[8] distcc distributed compiler. https://github.com/distcc/distcc.

[9] LibVPX: Vp8/vp9 codec sdk. https://www.webmproject.org/code/.

[10] Docker. https://www.docker.org.

[11] Eelco Dolstra. *The purely functional software deployment model*. Utrecht University, 2006.

[12] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. The Utility Coprocessor: Massively parallel computation from the coffee shop. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010.

[13] Stuart I. Feldman. Make – A Program for Maintaining Computer Programs. *Software – Practice and Experience*, 9(4):255–65, 1979.

[14] FFmpeg. https://github.com/FFmpeg/FFmpeg.

[15] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, 2017. USENIX Association.

[16] GIMP. https://www.gimp.org/.

[17] Google Test — Google Testing and Mocking Framework. https://github.com/google/googletest.

[18] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[19] Clark Allan Heydon, Roy Levin, Timothy P. Mann, and Yuan Yu. *Software Configuration Management Using Vesta*. Springer Publishing Company, Incorporated, 2011.

[20] Icecream distributed compiler. https://github.com/icecc/icecream.

[21] Inkscape, a powerful, free design tool. https://inkscape.org.

[22] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[23] Eric Jonas, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. *CoRR*, abs/1702.04024, 2017.

[24] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[25] The LLVM compiler infrastructure. http://llvm.org.

[26] Zhiqiang Ma and Lin Gu. The limitation of MapReduce: A probing case and a lightweight solution. In *Proc. of the 1st Intl. Conf. on Cloud Computing, GRIDs, and Virtualization*, pages 68–73, 2010.

[27] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[28] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, 2018. USENIX Association.

[29] OpenSSH. https://www.openssh.com.

[30] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. Scanner: Efficient video analysis at scale. In *ACM Transactions on Graphics*, 2018. Software available from https://github.com/scanner-research/scanner.

[31] Protocol Buffers. https://github.com/google/protobuf.

[32] Python. https://www.python.org.

[33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[34] George Sammons. *Learning Vagrant: Fast Programming Guide*. CreateSpace Independent Publishing Platform, USA, 2016.

[35] Ad Hoc Big Data Processing Made Simple with Serverless MapReduce. https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/.

[36] Apache Spark on AWS Lambda. https://github.com/qubole/spark-on-lambda.

[37] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[38] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[39] xkcd — Compiling. https://xkcd.com/303/.

[40] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.