A Thunk to Remember:
`make -j1000` (and other jobs) on functions-as-a-service infrastructure

Sadjad Fouladi   Dan Iter   Shuvo Chatterjee
Christos Kozyrakis   Matei Zaharia   Keith Winstein

Stanford University

Draft of Sep. 25, 2017 (under double-blind review; please do not redistribute)

Abstract

We present gg: a system for executing interdependent software workflows across thousands of short-lived "lambdas" that run in parallel on public cloud infrastructure. The system includes three major contributions: (a) an interchange format for representing "thunks"—programs and their complete data dependencies—that can be executed anywhere; (b) a system to automatically infer the dependency tree of a software build system and synthesize it as a directed acyclic graph of thunks, by replacing stages of the system with "models" that capture dependencies with fine granularity; and (c) an execution engine that resolves thunks recursively on public functions-as-a-service infrastructure with thousand-way parallelism.

We found that gg outperforms existing schemes for accelerating compilation—with large projects such as inkscape and LLVM, gg was 3.7× to 4× as fast as outsourcing compilation to a remote 64-core machine, and 1.2× to 2.2× as fast as running `make -j64` locally on the 64-core machine itself—and that the thunk abstraction is applicable to a broad range of tasks.

1 Introduction

The scale and elasticity of cloud platforms gives us an opportunity to dramatically rethink many computer applications. As cloud-computing platforms have developed, they have invariably become more fine-grained: for example, cloud vendors today offer hyper-elastic "serverless computing" platforms that can launch thousands of Linux containers within seconds [4, 21, 32], and all the major cloud vendors have switched to one-minute or ten-minute minimum billing increments for VMs [7, 18, 22]. Researchers have already taken advantage of these platforms to implement hyper-elastic versions of compute-intensive applications including video encoding [21] and MapReduce [32].

Making hyper-elastic platforms broadly available to more applications, however, will require *general* APIs and systems for accessing them, as opposed to the application-specific systems designed in prior work. To this end, we propose gg, a system for efficiently and easily capturing a parallelizable application consisting of a DAG of processes and executing it on serverless platforms.

Unlike previous work, gg can be applied to arbitrary workflows of Unix-like processes, such as software builds, scientific workflows, and video-processing pipelines. We first evaluate gg on one of the most challenging and well-studied parallelization problems: software builds. In this setting, we show that gg can extract *more* parallelism out of builds than traditional parallel and distributed build tools such as `make`, `distcc`, and `icecc` [16, 19, 29], while inferring the dependencies between build steps automatically, and that it performs up to 3.9× better than outsourcing to a long-running 64-core build server.

These gains are due both to increased parallelism (gg can exploit thousand-way parallelism from serverless infrastructure, and can infer parallelism opportunities not found in the original Makefile) and fewer network round trips, because gg keeps track of intermediate dependencies in the cloud and does not need to fetch intermediate build products back to the local machine. In addition, we show that gg can be used to implement video processing and MapReduce workloads similar to those in ExCamera [21] and PyWren [32].

To achieve these results, gg is built on two key concepts. First, gg represents work graphs through an abstraction called *thunks*, which are self-contained units of computation specifying both the executable to run and its dependencies. Second, gg provides a novel mechanism for automatically capturing an accurate dependency graph from an existing application, called *model substitution*. We briefly outline these concepts in turn.

Thunks: gg's thunks are self-contained units of computation that specify both an executable to run for a workflow step and its dependencies and environment. For example, in a build system, the thunk for compiling a single C source file will reference, as dependencies, the content hashes of the source file, the header files it includes as dependencies, and the compiler binary itself. In a video processing job, a thunk might specify one chunk of the video and the encoder binary that will operate on it.

Because thunks identify their complete functional footprint, gg can evaluate them in diverse environments, including a local sandbox or an AWS Lambda function. Each thunk is named canonically in terms of its computation task and dependencies (which can be other thunks),

which allows gg to memoize and reuse thunk results.

Model substitution: Although applications can specify a thunk graph directly, gg also provides a novel and easy-to-use mechanism for capturing an accurate dependency graph from an existing multi-process application, called model substitution. In this mechanism, gg can substitute *model programs* for each CPU-intensive executable invoked by a high-level driver process for the application (e.g., `make`, `cmake`, `ninja`, or even a shell script), by simply placing them in the `PATH`. As the driver process runs, these models execute only the minimal computation required to determine each execution's dependencies; for example, gg's model for the C linker determines which libraries will be consulted to produce the output file, but does not actually link them. Each model program then outputs a thunk representing that computation and its inputs. We found that models are an effective way to automatically and correctly infer dependencies from large legacy applications; for example, by including models for just **six** common executables (`gcc`, `g++`, `ld`, `ar`, `ranlib`, and `strip`), gg can automatically capture the dependency graphs of many large open-source projects.

We demonstrate gg through several example applications. First, to showcase a complex application that is not supported by previous "serverless" systems, we used gg to implement a parallel build accelerator. Builds have traditionally been challenging to parallelize efficiently for two reasons. First, to leverage parallelism, the build tool needs an accurate and *fine-grained* description of the dependencies (e.g., from a `Makefile`): overspecifying dependencies will reduce parallelism, while underspecifying them will lead to incorrect results. Second, current parallel build-outsourcing tools, such as `distcc` and `icecc` [16, 29], require many round trips between a master server and its workers (e.g., to send back intermediate build products), and perform poorly on a higher-latency connection to a public cloud. In contrast, using model substitution, gg automatically discovers a fine-grained but accurate dependency graph for a build simply by running the existing build system (e.g. `make`). This sometimes finds more parallelism than the build system has itself. Then, gg's execution engine allows it to upload all input files and submit the execution graph without repeated round trips, running with up to 1000-way parallelism.

1.1 Summary of results

gg on AWS Lambda outperforms existing parallel outsourced build systems running in the EC2 cloud, without requiring changes to the program's build system.

For example, compiling `inkscape` (a free-software illustration tool) requires 33.5 minutes on one core of a 4-core VM, and 5 minutes when using `icecc` to outsource builds to a separate 64-core VM in the same EC2 region. By contrast, gg can execute the same build system in 1.25 minutes on AWS Lambda, with each stage billed with subsecond granularity.

Compiling LLVM (a toolkit for writing compilers) requires 86 minutes on a single core, 4 minutes using `icecc` to outsource to a 64-core VM in the same region, and 1.2 minutes with gg.

These gains come with a caveat: automatically inferring dependencies in the first place requires running the original build system (e.g. make) with the compiler and other programs replaced with models. The build system itself can become a bottleneck, especially if it involves recursive make and many dependencies and the client machine is not well-endowed with CPU resources. On a cold start, inferring the tree of dependencies for `inkscape` required 2.5 minutes on the client machine (the 4-core VM), and 2.75 minutes for LLVM.

Apart from software builds, we also use gg to implement a video processing workload similar to ExCamera [21] and a MapReduce engine similar to PyWren [32], to show that gg's thunk abstraction is general enough to capture these applications.

In summary, we believe that the ability to divide common computational tasks into fine-grained tasks and execute them over hyper-elastic functions-as-a-service platforms like AWS Lambda, Google Cloud Functions, IBM OpenWhisk, and Azure Functions will become a new foundational use of these platforms. gg offers general and powerful mechanisms to achieve this for a common class of applications—multi-process Unix-like applications. gg is effective for a wide range of tasks ranging from "embarrassingly parallel" MapReduce to parallel builds with complex, irregular dependencies.

gg is open-source software; we have posted an anonymous version for review at https://github.com/gg-anon.

2 Related Work

gg is related to several different classes of systems.

Workflow systems. gg treats computations as a DAG of tasks, in a similar manner to Dryad [31], CIEL [39], Spark [50], and many scientific workflow systems [35]. However, gg differs from these systems in two important ways: its abstraction of a *thunk* to capture each task, and the manner in which gg can build up the thunk graph from an arbitrary (Unix-like) application using models.

gg's thunk abstraction differs from tasks in typical workflow systems in two ways. First, a thunk in gg is a self-contained process invocation with all of its dependencies captured, making it possible to take pieces of unmodified multi-process applications (e.g., build scripts) and execute them incrementally or in parallel on remote computing infrastructure. Second, higher-order thunks

Draft of Sep. 25, 2017 (under double-blind review; please do not redistribute)

allow gg to reference and analyze properties of derived results *without* running the full computation (e.g., deduplicating two references to the same higher-order thunk or running a model program that can process a thunk to determine the effect on a downstream computation).

gg's model programs, meanwhile, are a novel way to capture a dependency graph. By replacing each executable with a model that can take thunks as input and only determines which files a program invocation depends on, gg can identify dependencies at a fine granularity within existing applications, such as build scripts—sometimes unlocking *more* parallelism than the original application exposed. gg is not a build system itself and does not require rewriting or replacing mechanisms such as the `Makefile` or Bazel `BUILD` file; it seeks to work with existing software packages by automatically inferring dependencies from the existing build system, then outsourcing execution to serverless infrastructure.

Build systems and tools. Numerous build systems (including Vesta [28], `make` [19], and Bazel [8]) and outsourcing tools (such as `distcc` [16], `icecc` [29], and `mrcc` [37]) seek to incrementalize, parallelize, or distribute compilation to more-powerful remote machines. The state of the art generally has two limitations that gg tries to address: requiring a manually-specified dependency graph, which creates problems if users overspecify dependencies, and inefficient execution over a high-latency network (large number of round trips).

Virtually all existing build systems analyze a manually-specified configuration file, such as a `Makefile` or Bazel's `BUILD` script, to identify dependencies and enable incrementalization or parallelism. While this analysis can be highly sophisticated [27], misconfiguring the build will cause the system to run slower than necessary (if unneeded dependencies are given).

Such misconfiguration is not uncommon: for example, the popular `automake` tool processes subdirectories sequentially in order; in effect creating spurious dependency edges that reduce the available parallelism. gg is able to infer these dependencies accurately with fine granularity. As we show in Section 4, gg (with 64-way parallelism) outperforms `make -j64` (also 64-way parallelism) when compiling FFmpeg and Mosh, by about a factor of two in each case. The number of cores is the same, but gg is able to unlock more parallelism from the underlying Makefile.

gg does require model programs to be designed for every executable that will consume significant CPU resources, and every executable that will be used "downstream" to process the output of such programs, but the number of such executables is typically small (5–10), and they are shared across many applications.

In addition, many existing remote compilation systems, including `distcc` and `icecc`, send data between a master node and the workers frequently during the build, e.g., retrieving each intermediate file back to the master. These systems thus perform best on a local network, and add substantial latency when building on more remote servers in the cloud. In contrast, gg can upload *all* the build input once and have thunks execute and exchange data purely within the cloud, minimizing the impact of latency.

Caching of build products. `ccache` [9], whether alone or in concert with `distcc` or `icecc`, allows for cached compilations, reducing the time required for subsequent compiles of files that have not changed. `ccache`'s main limitation is that at present, it only caches single file compilations [9], so steps such as multiple file compilations, linking, or any other part of the build process are re-run. Bazel also caches build products, but it will not detect if a system include file has changed. In contrast, gg's memoization system is agnostic to the type of computation being performed and captures *all* dependencies, sufficient for diverse users to share a common cache (§3.4).

Incremental recomputation. Beyond build systems, automatic incremental computation has been proposed for data flow systems [24, 45, 50] and relational databases [10, 25, 34]. gg applies this technique to thunks representing arbitrary Unix-like processes and uses cloud storage to efficiently cache and reuse computation results.

Process migration and distributed OSes. gg's goal of transparently running a process on remote infrastructure is similar to that of distributed OSes [44, 48, 49], VM migration [12, 40] and container services [17]. Snowflock [33] combines VM cloning with distributed execution to accelerate parallel applications through a VM-fork abstraction and shows that this abstraction works efficiently under `distcc`. In gg, however, we chose to explicitly represent the computation graph up-front in order to minimize the number of network round trips and to enable incremental computation and caching.

Cloud computing. ExCamera [21] uses AWS Lambda to scale out video encoding and processing tasks over thousands of serverless function invocations, while PyWren [32] proposes a MapReduce implementation on AWS Lambda. In contrast, gg allows leveraging serverless computing platforms for a much broader set of workloads using the abstraction of thunks, including irregular execution graphs such as those that arise in a build system. Moreover, gg automatically enables incremental re-execution when only part of the input data or the program changes, as well as shared caching of input or intermediate files among multiple users. gg's thunks can also be used to express video coding tasks or MapReduce, as we show in Section 4.5.

Draft of Sep. 25, 2017 (under double-blind review; please do not redistribute)

3 Design and Implementation

Functions-as-a-service infrastructure presents a new parallel-processing substrate: services like AWS Lambda, Google Cloud Functions, and Azure Functions allow users to invoke thousands of parallel threads with subsecond startup times and subsecond billing. Although originally intended for asynchronous Web microservices, recent work has used such infrastructure to synchronously invoke thousands of threads at once, for tasks such as video encoding [21] and data analysis [32].

gg is designed as a general system for representing large workflows that can be executed on functions-as-a-service infrastructure. The expectation is that users will pursue a variety of "laptop extension" tasks, by taking a computation that might normally run locally for a long time (software compilation, interactive data exploration and visualization, machine learning, video encoding and filtering), and instead outsourcing it to thousands of short-lived parallel threads in the cloud, in order to achieve near-interactive completion time.

To explore this idea in concrete terms, we built a set of components that transform one type of CPU-intensive job (software compilation) into gg's representation, so that users can outsource builds to the cloud with thousand-way parallelism—the equivalent of "`make -j1000`". The following are the major components of gg:

1. The *thunk* abstraction for representing a morsel of deterministic computation in terms of a function (the hash of an x86-64 executable) and its complete functional footprint: the arguments, environment, and content hashes of all files it can access, some of which may be the output of other as-yet-unevaluated thunks. This component is agnostic to the environment where the thunk will be "forced" (meaning executed), and agnostic to the computation performed.

2. Software to *infer* a directed acyclic graph of interdependent thunks from an arbitrary software build system (whether `make`, `cmake`, `ninja`, `bazel`, etc.) that makes use of a C or C++ toolchain (compiler, linker, etc.). This component is specific to software compilation.

3. *Execution engines* for forcing a thunk, and all thunks that it depends on, recursively, in various environments: locally, outsourced to a remote computer, or on functions-as-a-service infrastructure. These engines also memoize thunk evaluation: they record a correspondence between the hash of a thunk and the hash of its output, and can short-circuit the later forcing of the same thunk.

In total, these components are implemented in about 8,800 lines of C++. We describe each in turn.

3.1 Thunks: units of delayed deterministic computation on data

In the functional-programming literature, a thunk is a parameterless closure (a function) that captures a snapshot of its environment for later evaluation [1]. The process of evaluating the thunk—which requires first evaluating any thunks that the thunk has captured, recursively—is known as "forcing" the thunk.

gg's goal is to outsource computations from the user's computer to thousands of functions in the cloud, in a way that guarantees that the user will get the same answer wherever the computation is run. To do this, we designed a concrete realization of a self-contained unit of computation—a thunk—in this context.

This representation has several design goals. First, it needs to identify all the information necessary to execute the computation. Because a cloud execution environment may be a significant distance from the user, we believe that simply exporting the user's filesystem to the cloud and allowing dependencies to be discovered dynamically, one by one, would require too many round trips compared with loading all dependencies upfront in one shot.

In addition, the thunk needs to be portable to a variety of computing environments and to different functions-as-a-service providers. It should identify inputs in a granular, cacheable, content-addressed way, because some referenced data may already be available in the cloud and some may need to be uploaded. For any particular execution on a particular dataset, there should be a canonical name for the thunk, so that the thunk can be memoized and need not be executed twice.

To satisfy these requirements, gg represents a thunk as a JSON document (Figure 1) that contains:

1. The SHA-256 hash of an x86-64 ELF binary executable using the Linux kernel ABI.
2. A list of arguments and environment variables that the executable will be invoked with.
3. The name of the output file. After the executable has finished running, the contents of this file will be considered the value of the thunk.
4. A list of "infiles"—the files that the program will access. These include data files accessed by the program, other executables invoked by it, and any shared libraries loaded at startup time. Each infile is identified by:
   (a) a name,
   (b) the SHA-256 hash of its contents,
   (c) the size in bytes, and
   (d) its "order."

An infile whose contents are known at the time the thunk is created will have order zero, and the SHA-256

Draft of Sep. 25, 2017 (under double-blind review; please do not redistribute)

```
{ "function": { "exe": "/__gg__/g++",
                "args": [ "-x", "cpp-output",
                          "preprocessed_output",
                          "-o", "compiled_output", "-S",
                          "-specs=/__gg__/gcc-specs" ],
                "hash": "sJ.rk55", },
  "infiles": [
   { "filename": "preprocessed_output",
     "hash": "KSwy9c",
     "order": 1, },
   { "filename": "g++",
     "hash": "sJ.rk55",
     "size": "1197328",
     "order": 0, },
   { "filename": "gcc-specs",
     "hash": "wYsT5z3",
     "size": "10381",
     "order": 0, },
   { "filename": "cc1",
     "hash": "LuVgfa3",
     "size": "23787992",
     "order": 0, } ],
  "outfile": "compiled_output" }
```

Figure 1: An example thunk, describing the compilation stage in a "Hello world" program, from preprocessed C to compiled assembly. The preprocessed C code (the first infile, hash starting with `KSwy9c`) is a thunk of order 1, making this a second-order thunk. SHA-256 hashes have been shortened for display, and some less-important fields have been omitted.

hash that identifies the file will be taken over its actual contents. The resulting thunk is considered a first-order thunk: a thunk that can immediately be executed because its infiles are all actual files. A thunk may also be of higher order, with one or more of its infiles taken as the output of other (as-yet-unevaluated) thunks. In this case, the SHA-256 hash is taken over the contents of the referenced thunk (the JSON document), and the order of the referencing thunk is one greater than the maximum order of any infile.

3.1.1 Execution

To force a thunk and retrieve its value, the execution environment follows several steps:

1. Retrieve the infiles, each one identified by a hash of its contents. These may come from any content-addressed storage (e.g. a directory or database where files are named by their hashes).
2. If the thunk is not a first-order thunk, then its non-zero-order infiles must first be forced recursively themselves. After this process completes, the original thunk is then rewritten so that the higher-order infiles are replaced with corresponding zero-order infiles (whose contents are the result of forcing the original infiles). This changes the contents of the overall thunk so that it is now a first-order thunk.
3. The first-order thunk can then be looked up in a cache (indexed by a hash of its contents) to see if its value has already been memoized.
4. If not, the executable is run with the provided arguments and environment variables—optionally in a sandbox that enforces the boundaries of the thunk and fails with an error if the executable attempts to reference data not included in the thunk.
5. The contents of the output file are taken as the value of the thunk.

We implemented the sandbox in about 800 lines of C++, using the `ptrace` Linux system call. It exits with an error if the program references any file not mentioned in the thunk. It also prevents the use of system calls such as `getpid`, `gettimeofday`[1], `getcpu`, and `socket`, which could be used to introduce outside information. The sandbox does not guarantee perfectly deterministic execution; an executable can still call system calls such as `brk`, which may return different results on different invocations. It is more like a unit test to gain confidence that the executable is not inadvertently accessing data outside the functional footprint specified in the thunk.

A thunk is somewhat akin to an operating-system container. Compared with Docker containers and similar schemes, gg's thunks are designed to be: (1) deterministic and content-addressed, by naming the executable and all its inputs by hash, (2) expressable in terms of other, lower-order thunks as input files, (3) runnable *within* a cloud function provided by functions-as-a-service infrastructure, and (4) factorable—some infiles (like the compiler or system libraries) may already be available in the cloud, and those infiles should not need to be re-uploaded every time the user wishes to force a thunk remotely.

3.2 Modeling compilation with thunks

Having designed a thunk abstraction to express the application of an x86-64 executable to named data, we next built a series of tools to infer a thunk for each output of a software build system. These tools allow gg to be used with most build systems: `make`, `cmake`, `ninja`, `bazel`, raw shell scripts, etc.

The basic approach is to model each stage of the compilation process with a program that understands the behavior of the underlying program just enough so that when the model is invoked, it can write out a thunk and quickly exit. The resulting thunk must capture the future dependencies of the underlying program with perfect recall (every file referenced must be mentioned in the thunk), or else forcing the thunk will fail. It should also capture these dependencies with good precision, or else the thunk will include unnecessary files that will need to be uploaded

[1] We also link executables with a modified version of libc that prevents use of the vDSO to get the time or CPU number without a syscall.

Draft of Sep. 25, 2017 (under double-blind review; please do not redistribute)

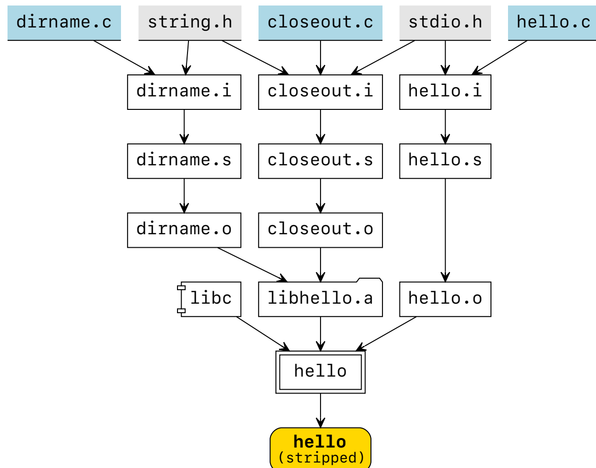

Figure 2: Part of the DAG of thunks inferred with `gg-infer` from the GNU `hello` example program. (Many system header files and other dependencies omitted to simplify the diagram.)

when forcing the thunk on a remote machine. When the thunk is created, all of the dependencies are captured and copied to the *object store*: a directory in the filesystem that contains each infile, named by the hash of its contents.

We built models for the GNU preprocessor, the `gcc` and `g++` compilers, the GNU assembler and linker, the archiver (`ar` and `ranlib`), and for ancillary tools like `strip`. Each of these models parses the command-line arguments for the underlying tool, and writes to the same output files that the real program would write to, but with a reference to a thunk instead of the true output.

At runtime, the user can extract the dependency tree for a software package by running `gg-infer` followed by the build command, e.g., `gg-infer make`. This tool runs the given command after changing the PATH environment variable so the models (e.g. for `gcc`) replace the underlying tools. An example DAG of thunks, inferred from the GNU `hello` example program, is shown in Figure 2.

Build systems often include scripts that run in addition to these standard tools (e.g. to generate configuration header files), but typically such scripts run "upstream" of the preprocessor, compiler, etc. These do not cause difficulty for our approach, because the scripts are simply run during the `gg-infer` step, and their output is then captured in a thunk (e.g. by the preprocessor).

However, `gg-infer` must handle the case where the build system tries to run or link against a thunk. Sometimes build systems do not simply run scripts: they first compile a helper program and then execute it. This could cause difficulties because if the file that should contain the helper program is a thunk, the program will not be able to be executed.

To deal with this case, our models instead write a thunk "reference" to whatever files would be written by the underlying tool. A thunk reference includes the hash of a particular thunk, but is also valid syntax for another kind of file. For example, the output of the linker is typically an executable, so the output of the linker *model* is a thunk reference that acts like an executable: it is a shell script that includes the hash of the thunk. If executed, it will force the thunk to compile the helper program and then execute it. When modeling a compiler stage that produces a library, the model writes a thunk reference that is a syntactically valid linker script, in case the build system tries to link against a build product.

In some cases, these references may not be sufficient; for example, a build system might seek to bundle compiled output into a ZIP file, or compare compiled output against a canonical reference and take action based on whether it matches. To handle this case, we built a wrapper tool called `gg-mock` that uses `ptrace` to detect if a "non-thunk-aware" program opens a thunk reference for reading; if so, it pauses execution of the thread in question until the thunk can be forced and the thunk reference can be replaced with the value of the thunk. Models indicate they are "thunk-aware" by trying to open a nonexistent file that has a canonical filename; `gg-mock` takes this as a signal to stop tracing the program.

3.2.1 Implementation details of models

Tools like `strip` and `ranlib` read a single file that is named on the command line and modify it in-place; these models are trivial.

Tools like `gcc` are more difficult. We model the compiler as a sequence of up to four stages: preprocessing, compiling, assembling, and linking. The `gcc` model parses the command-line options and determines which of the four stages will be executed. (Arguments like `-E`, `-S`, and `-c` cause the compiler to stop after preprocessing, compiling, or assembling.) The model constructs separate thunks for each stage, so that compiling two different source files may still result in a memoization hit if, for example, their preprocessed output is the same.

We torture-tested the `gcc` and `g++` models by running them against an array of free-software programs, including the `llvm` toolkit, applications like `inkscape`, `ffmpeg`, and OpenSSH, and the Linux kernel. These build systems use a wide variety of `gcc` options, which the models now handle.

To model the preprocessor, the thunk must include the full list of dependencies, including every file included with a `#include` directive, recursively. The model uses a feature of the real preprocessor (`gcc -M`) to obtain this list. The model locally caches the output of this command, keyed by the hashes of the contents of the referenced files. If none of those files has changed, the preprocessor model can safely reuse the previous dependency list. We use a

Draft of Sep. 25, 2017 (under double-blind review; please do not redistribute)

similar technique to model the linker.

The models also cache the hash of referenced files by inode number and nanosecond timestamp; if these have not changed, the models do not re-hash the input file.

3.2.2 Building a canonical toolchain

gg seeks to execute thunks efficiently either locally or in the cloud (on a remote VM or functions-as-a-service infrastructure). The thunk's functional footprint includes the hashes of data referenced by the executable (e.g. source code), but also the executable itself. We built canonical x86-64 versions of the GNU C/C++ toolchain that can run efficiently on the user's machine but will also be available in the cloud. This toolchain includes the following programs from GCC 7.2.0 and binutils 2.2.8: `ar`, `as`, `cc1`, `cc1plus`, `collect2`, `gcc`, `g++`, `ar`, `ranlib`, `ld`, `nm`, and `strip`.

We did not modify the tools, but we statically linked them against the `musl` minimalist standard C library to reduce their total size. Additionally, we modified `musl` to make the programs *thunk-aware*. At startup, the C library checks for an environment variable that contains the thunk, parses the thunk's list of infiles, and makes a correspondence table between names and hashes. Later, when the program calls `open` or another library function that references a filename, the C library replaces this with a reference to the file's true name (the SHA-256 hash of its contents). This layer of indirection allows these programs to become transparently thunk-aware, without modifying the filenames visible to the program, which would have affected debugging and diagnostic output.

We ensure the correctness of this indirection layer, and of the models, by compiling many projects two ways: normally with the toolchain, and by modeling the same steps and then forcing thunks inside the gg sandbox. gg's regression tests verify that both methods succeed and produce the same output.

3.2.3 Whole-tree optimizations of a DAG of thunks

By first extracting the entire chain of operations that produces the compiled output, gg's approach opens up the possibility of optimizations that apply to the whole DAG of thunks. For example:

1. Many build systems run `ranlib` to generate an archive index, but this step is a null operation if it comes after `ar s` (to create the archive)—which it often does. In that case, this stage can be safely skipped. (Currently, we do this optimization manually.)
2. The `-g` (debugging) flag can be eliminated from the compiler's command-line if the eventual output is going to be stripped.
3. Path subgraphs are linear chains of thunks, where the output of each thunk is the input to the next. These can be compressed into one thunk, or efficiently forced in the same cloud function, because the dependencies will already be available and will not have to be retrieved again from storage.

3.3 Forcing thunks locally or in the cloud

We built systems to force thunks in three environments: locally on a Linux computer, remotely to a computer running a persistent webserver, and on a commercial cloud-functions service (AWS Lambda). These systems, and the thunk abstraction, are agnostic to the process being executed and whether it is related to software compilation or not. The models in the previous section, however, are specific to software compilation.

The user starts with a thunk that represents the desired output of a large computation. If it is an executable thunk reference, they can simply execute it to force the thunk and then run the resulting program; otherwise they can run `gg-force` to explicitly force it. The method of forcing depends on which environment the user has selected.

3.3.1 Executing thunks locally

In this mode, `gg-force` reads the thunk in question and recursively builds the DAG of thunks that it depends on. These thunks are all identified by hash and located in the *object store* (the local directory that stores infiles named by hash). All first-order thunks can be executed in parallel. By default, gg only runs as many parallel thunks as the user has cores.

When each thunk completes, `gg-force` writes a memoization entry to the filesystem, recording that a thunk with hash x evaluated to an output of hash y. Any thunks that depended on the completed thunk x are rewritten to refer to y, and their order is recalculated (as one plus the maximum order of any infile). Any new first-order thunks can then be executed, and the process continues until the original thunk is executed.

3.3.2 Executing thunks remotely

When outsourcing computation, `gg-force` proceeds as above, with the exception that actual thunk execution happens on a remote computer (either a persistent computer or a cloud function). Before executing the thunk, gg needs to ensure that the thunk's infiles are available in the foreign environment. The system maintains a local representation of a *remote object store*—in our implementation, an Amazon S3 bucket. Every time it uploads a file (named by hash), it records this fact to the local object store so it will not need to re-upload it subsequently.

Draft of Sep. 25, 2017 (under double-blind review; please do not redistribute)

To remotely execute a thunk on AWS Lambda, `gg-force` first uploads any missing infiles to the remote object store, then invokes a lambda function with the thunk in the HTTP POST payload.[2] The lambda function looks at its local filesystem (which may contain infiles left over from a previous invocation), deletes any files that are not included as infiles to the current thunk, downloads any missing infiles from S3 (the remote object store), executes the function, and uploads the output to S3, named with its hash. The lambda function also responds, via HTTP, with the SHA-256 hash and size of the output. The contents of the output are not downloaded to the user's local machine; only the hash, so that the referencing thunks can be rewritten in terms of this output.

Remote execution on a persistent host is similar: we created a CGI script that implements the AWS Lambda interface and run it with the Apache webserver. When executing thunks on AWS Lambda, occasionally a thunk will require infiles bigger than the lambda's filesystem can accommodate. This typically happens during a final linking step; e.g., linking the `llvm` compiler requires more than two gigabytes of infiles, compared with a limit of 500 MB for the lambda's filesystem, which also must accommodate the output. In these cases, `gg-force` uses a persistent host as the backup "overflow" server. The reason for the "size" field in the thunk infiles structure is to allow `gg-force` to predict this situation in advance.

Implementation detail. We wrote routines in C++ to speedily upload and download files from S3 using several parallel HTTPS connections. Within each connection, these routines pipeline the HTTP requests (e.g. the uploaded files), uploading 32 files back-to-back without waiting for a response in between each one. We have found that S3 allows up to 100 pipelined requests in a row. This pipelining makes a considerable improvement in S3 upload and download throughput when transferring many files, even compared with Amazon's C++ SDK.

3.4 Collaborative use of gg

We envision that deployments of gg may *collaborate* in their caching of objects by hash and memoization that a particular thunk x evaluates to output y, so that no two users need run the same computation twice.

Many entities build a lot of software—e.g., continuous testing infrastructure like Travis-CI, and projects like Debian, Ubuntu, and Fedora. It would be convenient, and would further improve the speed of compilation, if no user needed to rebuild a file that had already been compiled once by one of these organizations. At the same time, it could be risky to blindly trust another entity's compiler output, in case the compiler has been compromised.

We expect that entities that execute thunks might publish two kinds of statements:

1. **Content-addressed storage assertions.** A cryptographically signed statement that, "The file with hash x has the following contents."

2. **Memoization assertions.** A cryptographically signed statement that, "The thunk with hash x has a value with hash y."

In both cases, a skeptical organization can double-check the assertion. If an entity makes a mistake, lies, or is compromised, it will be possible to prove to the world that it should no longer be trusted, because it will have issued a signed statement that is demonstrably false. One is unlikely to double-check *every* memoization assertion, because doing so is equivalent to simply forcing the thunk. We expect that users—even those interested in collaborative use of gg—will be more cautious in whom they trust to make memoization assertions vs. storage assertions.

4 Evaluation

To evaluate gg, we measured its start-to-finish build times under multiple scenarios with several unmodified open-source packages. We compared these times with existing build tools under the same scenarios.

Our findings show that for local builds, gg achieves either similar performance to the underlying build system, or in some cases (FFmpeg and Mosh), gg achieves significantly higher performance by inferring finer-grained information about dependencies, unlocking more parallelism. For distributed builds of large projects (Inkscape, LLVM) outsourced from a low-powered client (a 4-core EC2 VM) to the cloud, gg was 3.5× to 4× as fast as outsourcing compilation to a remote 64-core machine, and 1.2× to 3× as fast as running `make -j64` locally on the 64-core machine itself.

4.1 Evaluation set

To benchmark gg's performance and that of comparison schemes, we collected a selection of open-source programs written in C or C++. These packages cover a variety of build complexity, parallelization levels, and size. This set of programs consisted of `mosh` [38], `protobuf` [46], `llvm` [36], `ffmpeg` [20], `openssh` [42], `cmake` [13], and `inkscape` [30]. No changes were made to the underlying build system of these packages.

[2] We created different lambda functions for each combination of executables, so the executables would already be available when the lambda starts and would not need to be re-downloaded from S3.

Draft of Sep. 25, 2017 (under double-blind review; please do not redistribute)

4.2 Baselines

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

1, 2. **make, make (64):** In these two scenarios, the package's own build system was executed on a single core, or with up to 64-way parallelism (e.g. `make -j64`). For both, the build was done on a 64-core EC2 VM (`m4.16xlarge`) and no remote machines were involved.

3. **Icecream (64+1):** The package was built on a 4-core client (`m4.xlarge`), outsourcing to a 64-core VM in the same region. In order to achieve a fair comparison, the number of local jobs (executed on the master) was limited to one.[3]

4.3 gg's benchmarks

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

1. gg **(64):** The package was built locally using 64-way parallelism using gg.
2. `gg-remote` **(64):** With the same setup as the *Icecream (64+1)* experiment, the target package is built in a distributed fashion using gg. No jobs were executed on the client machine.
3. gg-λ **(64):** The build was initiated on a 4-core client machine, and thunks were executed on AWS Lambda with a maximum of 64 workers running at a time. A standby EC2 VM acted as the "overflow" worker for thunks whose total infile size was was too big for a lambda, as described in Section 3.3.
4. gg-λ **(1000):** The package was built on AWS Lambda using gg with 1000-way parallelism, again with a standby EC2 VM for thunks too large to execute on AWS Lambda.

We ran each scenario on each package at least 5 times. Figure 3 shows the median timings.

Cached builds. To evaluate the effectiveness of our caching mechanism, we compared gg against `ccache`. Starting with an empty cache, the package was built once to populate the cache. Then, after cleaning, the build was repeated again, this time on a "hot cache." Results were almost identical between gg and `ccache` (not shown because of space limits).

Dependency inference performance. As described in § 3, unlike the other approaches, gg separates the dependency inference and execution stages. Figure 4 shows the running times of `gg-infer` on a completely cold cache for each package, running the native build system through model substitution. These times are significant in the case of larger packages, and indicate that the build system machinery itself (e.g. recursive make), as well as the time required to hash the contents of all build dependencies to capture them in thunks, can consume a significant amount of CPU and I/O.

The overhead is less severe in the case of an incremental build, or without a cold cache, because gg caches the hashes and dependencies of files on disk and the underlying build system will not attempt to re-build every file. However, it does suggest a significant weakness to gg's approach of attempting to retain compatibility with the underlying build system, rather than asking software authors to discard `make` or `cmake` in favor of newer, more efficient build systems (e.g. Bazel). In future work, we plan to make it possible to adapt inferred thunks into a "build system template" that can be distributed with or alongside these open-source packages.

4.4 Discussion of evaluation results

In building `mosh` and FFmpeg locally, gg is roughly 2× faster than `make` running with the same degree of available (64-way) parallelism. `make`'s performance is hindered by spurious dependency edges. This indicates that gg's fine-grained approach is effective in reducing the build time for packages that consist of many small modules. Some build systems, like `automake`, generate separate build recipes for each module, and they build each module one after another, even though there may be no dependencies between each of these modules. By extracting the real dependencies, gg is capable of extracting more parallelism than the original build system.

Small packages. Although AWS Lambda allows the user to execute thousands of jobs at once, smaller software packages usually do not have this degree of parallelism. Nonetheless, these programs can still benefit from other gg features, such as global caching.

Figure 5 shows the timings from a `mosh` build using gg on AWS Lambda. For the preprocess stage, most of the workers' time is spent on downloading the dependencies—preprocessing a single file may depend on a few hundred header files. Because preprocessing happens for all the files simultaneously, its effect on the start-to-finish build time is negligible.

Larger packages. The benefits of using gg in building larger software packages are more significant. As an example, `ffmpeg` has about 1500 files that require compilation; given enough cores, almost all of that compilation can happen in parallel. Figure 6 shows the thunk start and end timings for `ffmpeg` and indicates that at a certain point, increasing parallelism yields diminishing returns;

[3] `icecc` does preprocessing and linking locally, so it is not possible to fully outsource the build job.

Draft of Sep. 25, 2017 (under double-blind review; please do not redistribute)

	Local			Distributed			
	make	make (64)	gg (64)	Icecream (64+1)	gg-remote (64)	gg-λ (64)	gg-λ (1000)
Mosh	29s	11s	05s	12s	12s	10s	12s
OpenSSH	36s	02s	03s	06s	27s	13s	15s
CMake	04m 31s	17s	15s	28s	49s	33s	25s
Protobuf	05m 26s	20s	22s	23s	41s	40s	33s
FFmpeg	09m 45s	40s	23s	01m 36s	01m 36s	59s	35s
Inkscape	33m 35s	01m 32s	01m 46s	05m 01s	02m 25s	02m 53s	01m 15s
LLVM	1h 16m 18s	02m 33s	02m 21s	04m 06s	03m 29s	02m 35s	01m 11s

Figure 3: Comparison of build times in different scenarios described in § 4.1.

	4 cores	64 cores
Mosh	02s	02s
OpenSSH	05s	03s
CMake	15s	09s
Protobuf	12s	03s
FFmpeg	01m 28s	36s
Inkscape	02m 33s	44s
LLVM	02m 45s	49s

Figure 4: Time required for gg to generate the thunks that it will execute later, either locally or remotely.

the last four jobs are primarily serial (archiving, linking, and stripping) and consume almost half of the total job completion time.

Figure 3 demonstrates that for very large packages, a high degree of parallelism can significantly improve build times, but that serial stages continue to dominate. gg compiles `inkscape` in 1 minute, 15 seconds on AWS Lambda with a cold cache, compared with 5 minutes when outsourced to a 64-core VM in the same EC2 region. This is a speedup of 4×. In a truly cold client where `gg-infer` must be run anew on the 4-core client to extract the dependency graph from the `inkscape` build system, the total time to build with gg (combining inference and execution) would be 3m48 (2m33 + 1m15), about 32% faster than outsourcing to the nearby 64-core VM.

Results for LLVM are similar, even though the single-core build time for this project is more than double that of `inkscape`. This indicates that serial steps (e.g. linking) dominate the job completion time, as in Figure 6.

4.5 Extending gg to other applications

4.5.1 Word frequency with MapReduce

Word frequency is a classic MapReduce problem [15] and the goal is to count the number of times that each word has appeared in a corpus of documents. Previous work has implemented such workloads on functions-as-a-service infrastructure [32]. To implement this workflow using gg, we defined the following set of functions:

`map`(i`.txt`) → i`.mapped`**:** the first line of map output is a list of offsets for partition program.

`partition(1.mapped,...,N.mapped)` → `P`i**:** partition program read the first line of the infile to find the range that it should read.

`reduce(P`i`)` → `R`i**:** reduce function goes through the infile and sums up all of the values for each key.

`cat(R1,R2,R3)` → `out`**:** concatenates the infiles together to create the outfile.

Figure 7 shows the dataflow for word fequency problem with N input files and 3 reduce workers.

4.5.2 Video encoding

Previous work has used functions-as-a-service infrastructure to run interdependent video processing tasks with many-way parallelism [21]. To demonstrate the expressive power of gg, we also implemented this scheme in terms of thunks. The functions necessary to encode a "batch" of video chunks were:

`vpxenc`$(U) \rightarrow C$**:** encoding an uncompressed video to a compressed VP8 video.

`xcdec`$(C,S) \rightarrow S'$**:** extracting the final state of a compressed video starting with a state.

`xcenc`$(U,C,S) \rightarrow Y$**:** re-encoding the first frame of an compressed video, starting with a state.

`rebase`$(U,C,S,S') \rightarrow Y'$**:** rebasing a compressed video starting with a state, on top of another state.

In summary, the algorithm first encodes each chunk in parallel using `vpxenc()` and then, in a serial process, rebases each output on top of the state left by the previous chunk using `rebase()` function. Figure 8 shows the dependency graph for encoding a batch of 4 chunks using these functions.

Draft of Sep. 25, 2017 (under double-blind review; please do not redistribute)

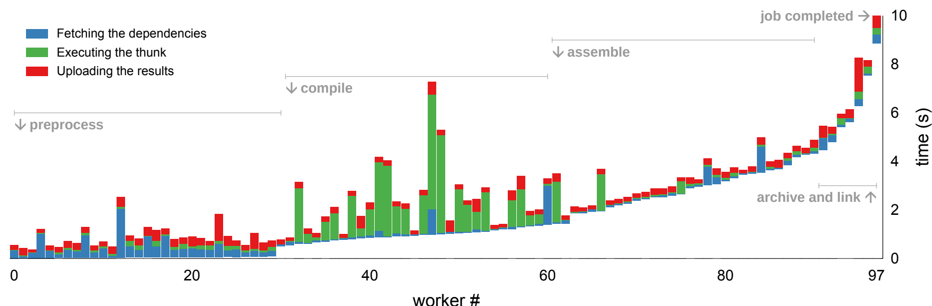

Figure 5: Breakdown of workers' execution time when building `mosh` using gg on AWS Lambda.

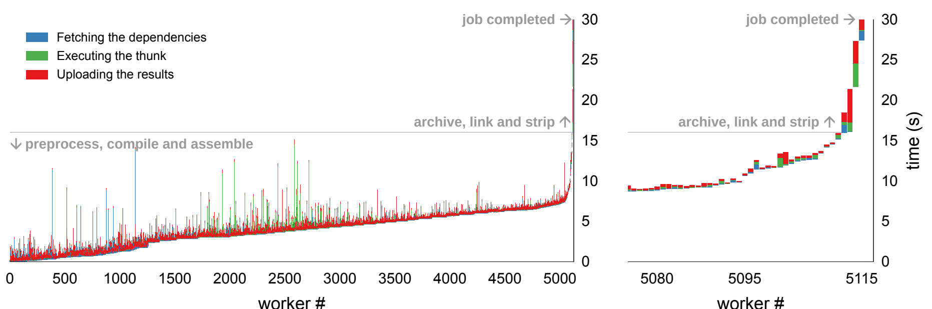

Figure 6: Breakdown of workers' execution time when building `ffmpeg` using gg on AWS Lambda. Serial stages (archiving, linking, and stripping) consume almost half the total job completion time. The right-hand plot shows a zoomed-in version of the figure.

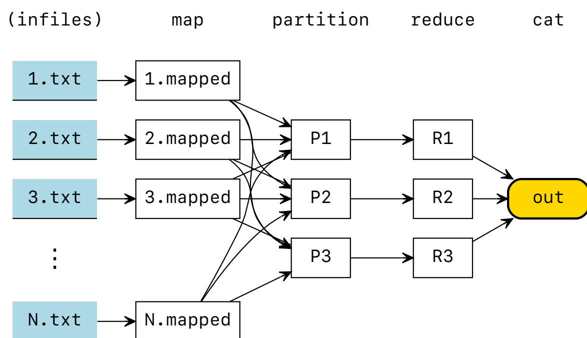

Figure 7: Dependency graph for a MapReduce workflow calculating word frequency, expressed in gg thunks.

5 Limitations and future work

gg has a number of important limitations that will be the subject of future work.

The underlying build system is a bottleneck. gg infers dependencies from the underlying build system (`make`, `cmake`, `bazel`, etc.) by running it under model substitution to generate thunks. In cases where the client is weak in comparison with available infrastructure to force thunks, this step can be a significant bottleneck. For example, on a totally cold cache, it took 88 seconds on a 4-core server to infer the dependencies from FFmpeg, compared with only 35 seconds to actually compile all of FFmpeg with 1000-way parallelism in AWS Lambda. This suggests a significant weakness to gg's approach of attempting to retain compatibility with the underlying build system. In future work, we plan to make it possible to adapt inferred thunks into a "build system template" that can be distributed with or alongside these open-source packages.

Often, tests are what is slow. gg's modeling approach only infers dependencies for programs it knows about: the compiler, linker, archiver, etc. This is sufficient to *build* a wide range of open-source programs, but often it is the testing of these programs, not the compilation, that is the true bottleneck. gg does not have an approach to automatically infer the dependencies of such tests; developers would need to manually create thunks to allow these processes to be outsourced.

Some build systems want to read intermediate data. Build systems like the Linux kernel try to read intermediate files and, in scripts, compare them against a canonical reference or check if they are empty. These builds must be run under `gg-mock`, which pauses execution of any "non-thunk-aware" thread that tries to read a thunk reference on disk. However, performance of the resulting build is poor, because thunks end up forced one-at-a-time, as the build system gets to each in turn. We will need to implement a system of intelligently pre-fetching thunks so they can be

Draft of Sep. 25, 2017 (under double-blind review; please do not redistribute)

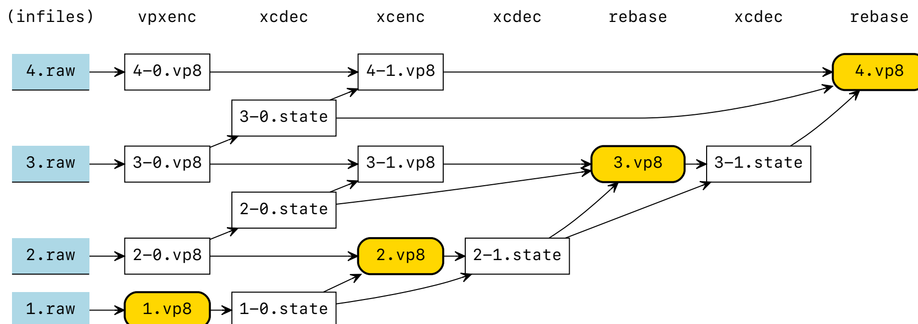

Figure 8: Dependency graph for a video-processing workflow [21], expressed in gg thunks.

forced in parallel, or persuade these developers to modify their build systems.

5.1 Future work in serverless services

Developing and using gg on AWS Lambda has led us to several observations about improving serverless services [4, 6, 23] and frameworks [41, 43, 47] for this application domain.

Storage. AWS Lambda functions can access long-term storage and database services, such as AWS S3 and DynamoDB [2, 5]. The scalability and durability of these services makes them appropriate for the input and output data of serverless computations. The missing component for applications like gg is *ephemeral storage* that allows communication of intermediate data between functions. The scalability, speed, and pricing for capacity and IOPS should match the characteristics of serverless computation, and referenced objects should automatically be garbage-collected when a job terminates.

Scheduling. Lambda functions are currently scheduled first based on the availability of processor and memory resources and next based on code locality (container already available or running on a host) [26]. For applications like gg, it is also important to consider *data locality*. The scheduler should steer each task towards functions that store or have fast access to its input data—or at least, a large set of the input data. The explicit data dependencies encoded in thunks allows for an intelligent scheduler that could match tasks to their data, prefetch data, or trade off communication overheads to the cost of waiting for resources to become available in a container with data.

Safety. As we expand the scope of lambda uses, it is important to provide mechanisms that constrain what a single lambda or an ensemble of lambdas can do, with whom they can communicate, and what resources they can consume and for how long (i.e., how much money they can spend and over how much time). This may involve using and extending existing mechanisms, both local like `cgroups` and pledges [11, 14], or cluster-wide like AWS IAM and limits [3].

Multiple applications and users. It will be useful to develop techniques that allow multiple applications and multiple users to share a fixed set of lambda resources. Current approaches such as quotes on the number of concurrently active lambdas can lead to deadlock among contending applications, and failures. It may be necessary to introduce priority and admission-control mechanisms for lambda-based frameworks like gg.

6 Conclusion

In this paper, we described gg, a system for executing interdependent software workflows across thousands of short-lived functions that run in parallel on public cloud infrastructure. We demonstrated gg's utility and performance benefits in compiling large software projects with thousand-way parallelism, and we expressed prior work in general-purpose serverless computing [21, 32] in terms of gg's thunk abstraction, so it can be run by the same execution engines.

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 today non-graphics GPGPU applications are a major use of GPUs: physical simulations, database queries, and especially neural networks and deep learning.

Cloud functions may tell a similar story. Although intended for asynchronous Web microservices, we believe that with sufficient systems, the same infrastructure is capable of broad and exciting new applications. Just as GPGPU computing did a decade ago, general-purpose lambda computing may have far-reaching effects.

Draft of Sep. 25, 2017 (under double-blind review; please do not redistribute)

References

[1] ABELSON, H., AND SUSSMAN, G. J. WITH SUSSMAN, J. *Structure and Interpretation of Computer Programs*, 2nd editon ed. MIT Press/McGraw-Hill, Cambridge, 1996.

[2] AWS DynamoDB. https://aws.amazon.com/dynamodb/.

[3] AWS IAM. https://aws.amazon.com/iam/.

[4] AWS Lambda. https://aws.amazon.com/lambda.

[5] Amazon S3. https://aws.amazon.com/s3/.

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

[7] Azure pricing overview. https://azure.microsoft.com/en-us/pricing/.

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

[9] ccache - a fast C/C++ compiler cache. https://ccache.samba.org.

[10] CERI, S., AND WIDOM, J. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1991), VLDB '91, Morgan Kaufmann Publishers Inc., pp. 577–589.

[11] cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[12] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2* (Berkeley, CA, USA, 2005), NSDI'05, USENIX Association, pp. 273–286.

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

[14] DE RAADT, T. Pledge: a new mitigation mechanism. https://www.openbsd.org/papers/hackfest2015-pledge/.

[15] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

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

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

[18] Per-Second Billing for EC2 Instances and EBS Volumes. https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes.

[19] FELDMAN, S. I. Make – A Program for Maintaining Computer Programs. *Software – Practice and Experience 9*, 4 (1979), 255–65.

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

[21] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. 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)* (Boston, MA, 2017), USENIX Association, pp. 363–376.

[22] Google Compute Engine pricing. https://cloud.google.com/compute/pricing.

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

[24] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 75–88.

[25] HALEVY, A. Y. Answering queries using views: A survey. *The VLDB Journal 10*, 4 (Dec. 2001), 270–294.

[26] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with openlambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2016), HotCloud'16, USENIX Association, pp. 33–39.

[27] HEYDON, A., LEVIN, R., AND YU, Y. Caching function calls using precise dependencies. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2000), PLDI '00, ACM, pp. 311–320.

[28] HEYDON, C. A., LEVIN, R., MANN, T. P., AND YU, Y. *Software Configuration Management Using Vesta*. Springer Publishing Company, Incorporated, 2011.

Draft of Sep. 25, 2017 (under double-blind review; please do not redistribute)

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

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

[31] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 59–72.

[32] JONAS, E., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the cloud: Distributed computing for the 99%. *CoRR abs/1702.04024* (2017).

[33] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A. M., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. Snowflock: Rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, 2009), EuroSys '09, ACM, pp. 1–12.

[34] LEE, K. Y., SON, J. H., AND KIM, M. H. Efficient incremental view maintenance in data warehouses. In *Proceedings of the Tenth International Conference on Information and Knowledge Management* (New York, NY, USA, 2001), CIKM '01, ACM, pp. 349–356.

[35] LIU, J., PACITTI, E., VALDURIEZ, P., AND MATTOSO, M. A survey of data-intensive scientific workflow management. *J. Grid Comput. 13*, 4 (Dec. 2015), 457–493.

[36] The LLVM compiler Infrastructure. http://llvm.org.

[37] MA, Z., AND GU, L. The limitation of mapreduce: A probing case and a lightweight solution. In *Proc. of the 1st Intl. Conf. on Cloud Computing, GRIDs, and Virtualization* (2010), pp. 68–73.

[38] mosh - Mobile Shell. https://www.mosh.org.

[39] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 113–126.

[40] NELSON, M., LIM, B.-H., AND HUTCHINS, G. Fast transparent migration for virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 25–25.

[41] Openlambda. https://github.com/open-lambda/open-lambda.

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

[43] Apache Openwhisk. https://github.com/apache/incubator-openwhisk.

[44] OUSTERHOUT, J. K., CHERENSON, A. R., DOUGLIS, F., NELSON, M. N., AND WELCH, B. B. The sprite network operating system. *Computer 21*, 2 (Feb. 1988), 23–36.

[45] POPA, L., BUDIU, M., YU, Y., AND ISARD, M. Dryadinc: Reusing work in large-scale computations. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2009), HotCloud'09, USENIX Association.

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

[47] Serverless framework. https://github.com/serverless/serverless.

[48] TANENBAUM, A. S., AND MULLENDER, S. J. An overview of the amoeba distributed operating system. *SIGOPS Oper. Syst. Rev. 15*, 3 (July 1981), 51–64.

[49] TANENBAUM, A. S., VAN RENESSE, R., VAN STAVEREN, H., SHARP, G. J., AND MULLENDER, S. J. Experiences with the amoeba distributed operating system. *Commun. ACM 33*, 12 (Dec. 1990), 46–63.

[50] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. 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* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.

Draft of Sep. 25, 2017 (under double-blind review; please do not redistribute)