Build in the Cloud: Distributing Build Steps

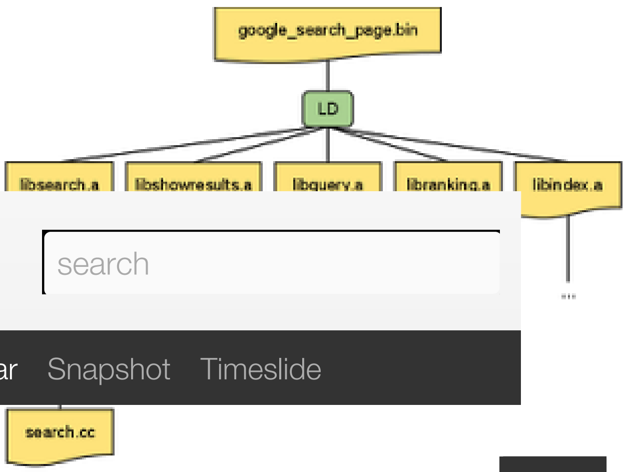
This is the third in a four part series describing how we use the cloud to scale building and testing of software at Google. This series elaborates on a presentation [http://www.youtube.com/watch?v=b52aXZ2yi08] given during the Pre-GTAC 2010 [http://sites.gtac.biz/gtac2010/] event in Hyderabad. Please see our first post [http://google-engtools.blogspot.com/2011/06/build-in-cloud-accessing-source-code.html] in the series for a description how we access our code repository in a scalable way. To get a sense of the scale and details on the types of problems we are solving in Engineering Tools at Google take a look at this post [http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html] .

The two previous posts described a custom file system for providing access to source code hosted in the cloud, and how the build system works. This post builds on these to describe a system for efficiently distributing build steps across a large number of machines. As you will see, details of the source file system and the build system are important to how we achieve fast and efficient distributed builds. So before we get into the mechanics of how build steps are distributed, it is helpful to call attention to a few points.

First, the build system is **content-based** where the internal bookkeeping for inputs and outputs is based on content digest, not file and timestamp (as in Make). This means that content equality is determined by comparing content digests, and these digests are tracked internally by the build system as it builds and operates on the action graph. Computing content digests for builds with a large amount of source code could be expensive because of the time spent reading files. We avoid this problem by computing and storing digests as content is checked in, and then providing digests directly to the build system as extended attributes [http://en.wikipedia.org/wiki/Extended_file_attributes] .

Second, the build system reads BUILD files to construct a graph of dependencies, and then uses the dependency graph to construct a graph of **build actions** or build steps. The graph is constructed by using the outputs of actions as inputs to other actions. Dependencies are expected to be completely specified and there is no dynamic dependency detection. The combination of content digests along with completely specified dependencies means that actions can be expressed as functions. In this **functional model**, the function inputs are the content digests and the environment (environment variables, command line options), the function is the tool or script which transforms inputs, and the outputs are the function results. These functional build actions are atomic units of work, and the transformation of inputs to outputs is opaque to the system. This means that build actions are language and tool agnostic and, among other things, may compile a C++ translation unit, compile Java files, link a binary, or even run a unit test.

An example of the action graph. Outputs of actions, such as `search.o` from from the `CC` action, become inputs to other actions (`LD` in this case).



Google Engineering T…

search

Classic Flipcard Magazine Mosaic Sidebar Snapshot Timeslide

Bug Prediction at … 46

Build in the Cloud:… 3

Trying on the new… 1

Build in the Cloud: Distrib…

Build in the Cloud:… 12

Are SSDs a silver … 10

Build in the Cloud:… 8

Testing at the spe… 31

C++ at Google: H… 12

Welcome to the G… 10

[http://4.bp.blogspot.com/-JQI9_NeFHqc/Tnu0zAbjXbI/AAAAAAAAABU/D2x5cr9ErHY/s1600/BuildintheCloudPart2HowtheBuildSystemworks+%25281%2529.png]

Third, we require that every build action is a function of only its declared inputs. This means that if we would execute the **same** action with the **same** input again we would get **bit-identical** output. This guarantees that not re-executing an action because the input files have the same content as in a previous build will not change the build results. This sounds like a reasonable requirement, but in practice an action may depend on things other than the declared input files, such as the contents of system header files or even the time of day (consider the __DATE__ macro expanded by the C preprocessor, or the timestamps embedded in every jar file). We address these problems by configuring our tools to make actions **hermetic** and post-processing some file types to overwrite timestamps.

Now let’s continue with details of how we use these functional actions to distribute the build.

As self-contained, atomic units of work, build actions are **portable** in that they can be sent to other machines, along with the inputs, for remote execution. This is useful at Google because we happen to have a lot of machines in data centers, which means we can distribute execution of build actions across thousands of workers. In this model all actions can be distributed, and the parallelism of the build is limited only by the width of the action graph.

Distributing actions across lots of workers makes the build fast, but we also found that workers duplicated a lot of work since many engineers build the same code. The functional nature of build actions, where the same inputs alway produce bit-identical outputs, means that we can easily and correctly cache and reuse results. We construct a cache key by computing a digest from the entire request (the command line and inputs), so it is not possible to “leave out” something and get incorrect cache hits. Remember that input files are described by content digest, so computing the cache key is relatively cheap, even for a large amount of content. When an action is ready for remote execution, we first compute the cache key. In case of a cache miss, the action is executed on a worker and the result is cached as it is returned to the user. In case of a cache hit, we simply return the cached result. To maintain the illusion that a cached action is actually executed, we also cache and replay the stdout+stderr of the action.

As changes are submitted to the code base, the first build to encounter each change waits a bit longer on actions affected by the code change because these need to be re-executed, but because build actions are distributed the difference is not very noticeable. In many cases, such as whitespace or comment changes in C++, different inputs still produce bit-identical output. Because the build system is content-based, this situation will then produce cache hits for subsequent actions, which provides another form of build avoidance. The end result is a **greater than 90% cache hit rate** overall. This means that even “clean” rebuilds are mostly reusing work done for previous builds, resulting in extremely fast build times. Another way to think of this is that each change to the code base results in an on-demand binary release of every library and executable affected by the change.

Distributing build load across many machines and reusing build actions have been so successful in speeding up builds that we inadvertently created another issue. A clean build of a large project may produce several gigabytes of output, and since these builds typically take less than a few minutes and we do tens of thousands of builds per day, the volume of data produced by the distributed build resulted in considerable load on our networks and local disk I/O. Our next and final post in this series will describe how we solved this problem.

Stay tuned! In this post we explained how builds are distributed and cached to make the build fast and efficient. Part four in this series describes how we solved the problem of dealing with the large amount of data produced by doing many builds very quickly.

- Nathan York

Posted 23rd September 2011 by Engineering

0 Add a comment

Enter your comment...

Comment as: Google Account

Publish Preview