Build in the Cloud: How the Build System works

This is the second 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] *.*

As you have learned in the previous post [http://google-
e-code.html] , at
ead. In this article
build system about
icle we will explain
ledge to distribute
the builds at Google across a large cluster of machines and share build results between developers.

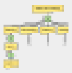
Google Engineering T... search

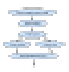
Classic Flipcard Magazine Mosaic Sidebar Snapshot Timeslide

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

10

From some of the feedback we got for our first p is clear that you want to see how we actually go about describing the dependencies we use to drive our build and test support.

At Google we partition our source code into smaller entities that we call **packages**. You can think of a package as a directory containing a number of source files and a description of how these source files should get translated into build outputs. The files describing that are all called `BUILD` and the existence of a `BUILD` file in a directory makes that directory into a package.

All packages reside in a single file tree and the relative path from the root of that tree to the directory containing the `BUILD` file is used as a globally unique identifier for the package. This means that there is a 1:1 relationship between package names and directory names.

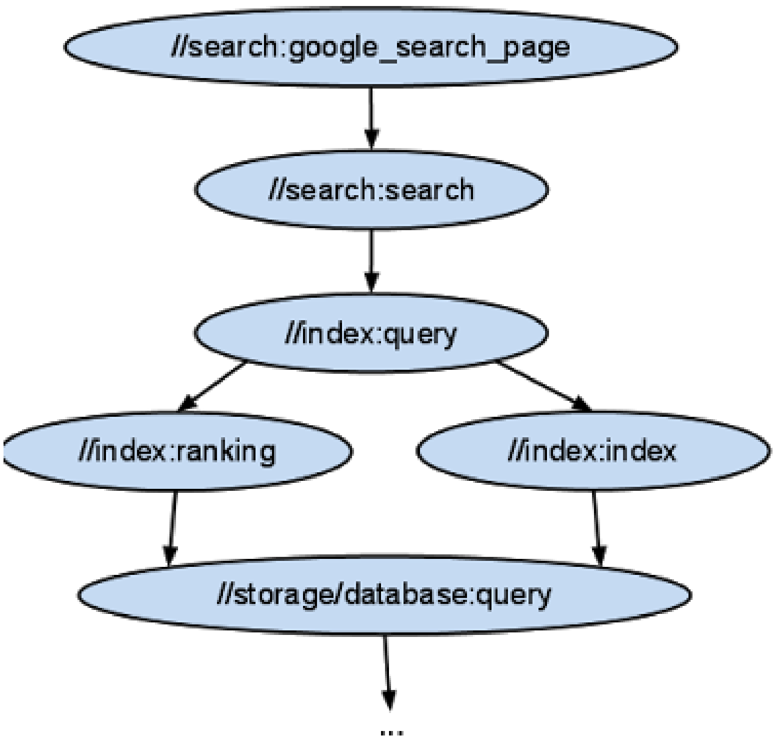
Within a `BUILD` file we have a number of named **rules** describing the build outputs of the package. The combination of package name and rule name uniquely identifies the rule. We call this combination a **label** and we use labels to describe dependencies across rules.

Let’s look at an example to make this a little more concrete:

```
/search/BUILD:
cc_binary(name = ‘google_search_page’,
          deps = [ ‘:search’,
                   ‘:show_results’])

cc_library(name = ‘search’,
           srcs = [ ‘search.h’,‘search.cc’],
           deps = [‘//index:query’])

/index/BUILD:

cc_library(name = ‘query’,
                  srcs  =  [  ‘query.h’,  ‘query.cc’,
‘query_util.cc’],
           deps = [‘:ranking’,
                   ‘:index’])

cc_library(name = ‘ranking’,
           srcs = [‘ranking.h’, ‘ranking.cc’],
           deps = [‘:index’,
                   ‘//storage/database:query’])

cc_library(name = ‘index’,
           srcs = [‘index.h’, ‘index.cc’],
           deps = [‘//storage/database:query’])
```

This example shows two `BUILD` files. The first one describes a binary and a library in the package `//search` and the second one describes several libraries in the package `//index`. All rules are named using the `name` attribute, You can see the dependencies expressed in the `deps` attribute of each rule. The ‘`:`’ separates the package name from the name of the rule. Dependencies on rules in a different package start with a “`//`” and dependencies on rules in the same package can just omit the package name itself and start with a ‘`:`’. You can clearly see the dependencies between the rules. If you look carefully you will notice that several rules depend on the same rule `//storage/database:query`, which means that the dependencies form a directed graph. We require that the graph is acyclic, so that we can create an order in which to build the individual targets.

Here is a graphical representation of the dependencies described above:

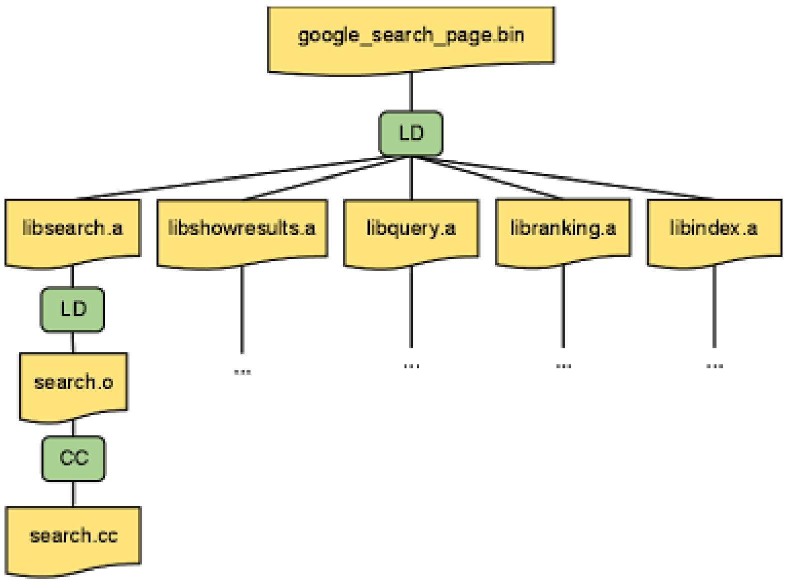


[http://1.bp.blogspot.com/-J33oNoBeNZQ/Tkz2d-FD9NI/AAAAAAAAABI/TKDHLsBd0qY/s1600/BuildintheCloudPart2HowtheBuildSystemworks.png]

You can also see that instead of referring to concrete build outputs as typically happens in `make` based build systems, we refer to abstract entities. In fact a rule like `cc_library` does not necessarily produce any output at all. It is simply a way to logically organize your source files. This has an important advantage that our build system uses to make the build faster. Even though there are dependencies between the different `cc_library` rules, we have the freedom to compile all of the source files in arbitrary order. All we need to compile `'query.cc'` and `'query_util.cc'` are the header files of the dependencies `'ranking.h'` and `'index.h'`. In practice we will compile all these source files at the same time, which is always possible unless there is a dependency on an actual output file of another rule.

In order to actually perform the build steps necessary we decompose each rule into one or more discrete steps called **actions**. You can think of an action as a command line and a list of the input and output files. Output files of one action can be the input files to another action. That way all the actions in a build form a bipartite graph [http://en.wikipedia.org/wiki/Bipartite_graph] consisting of actions and files. All we have to do to build our target is to walk this graph from the leaves - the source files that do not have an action that produces them - to the root and executing all the actions in that order. This will guarantee that we will always have all the input files to an action in place prior to executing that action.

An example of the bipartite action graph for a small number of targets. Actions are shown in green and files in yellow color:



[http://4.bp.blogspot.com/-wWe0mldN7aM/Tkz2eJO199I/AAAAAAAAABQ/jsHZhMcbHLU/s1600/BuildintheCloudPart2HowtheBuildSystemworks%2B%25281%2529.png]

We also make sure that the command-line arguments of every action only contain file names using relative paths - using the relative path from the root of our directory hierarchy for source files. This and the fact that we know all input and output files for an action allows us to easily execute any action remotely: we just copy all of the input files to the remote machine, execute the command on that machine and copy the output files back to the user's machine. We will talk about some of the ways we make remote builds more efficient in a later post.

All of this is not much different from what happens in `make`. The important difference lies in the fact that we don't specify the dependencies between rules in terms of files, which gives the build system a much larger degree of freedom to choose the order in which to execute the actions. The more parallelism we have in a build (i.e. the **wider** the action graph) the faster we can deliver the end results to the user. At Google we usually build our solutions in a way that can scale simply by adding more machines. If we have a sufficiently large number of machines in our datacenter, the time it takes to perform a build should be dominated only by the **height** of the action graph.

What I describe above is how a **clean build** works at Google. In practice most builds during development will be **incremental**. This means that a developer will make changes to a small number of source files in between builds. Building the whole action graph would be wasteful in this case, so we will skip executing an action unless one or more of its input files change compared to the previous build. In order to do that we keep track of the content digest of each input file whenever we execute an action. As we mentioned in the previous blog post [http://google-engtools.blogspot.com/2011/06/build-in-cloud-accessing-source-code.html] , we keep track of the content digest of source files and we use the same content digest to track changes to files. The FUSE filesystem described in the previous post [http://google-engtools.blogspot.com/2011/06/build-in-cloud-accessing-source-code.html] exposes the digest of each file as an extended attribute. This makes determining the digest of a file very cheap for the build system and allows us to skip executing actions if none of the input files have changed.

Stay tuned! In this post we explained how the build system works and some of the important concepts that allow us to make the build faster. Part three in this series describes how we made remote execution really efficient and how it can be used to improve build times even more!

- Christian Kemper

Posted 18th August 2011 by Engineering

12 View comments

ovidiu August 18, 2011 at 11:44 AM

Any chance of open-sourcing Blaze?

Reply

Replies

Adam MacBeth April 16, 2012 at 1:14 PM

This would be a huge boon to the community. Please consider!

Reply

duff August 18, 2011 at 11:49 AM

The important difference [between what happens in make] lies *in the fact that we don't specify the dependencies between rules in terms of files.*

I don't understand this claim. Are you saying that a traditional Makefile cannot be generated from your BUILD description files which exactly represents the bipartite action graph and thereby achieves the same level of parallelism?

Reply

Gilbert Chen August 18, 2011 at 12:51 PM

So the dependencies have to be specified explicitly in the build files. I can see two problems immediately comming from this approach:

1) you never know for sure if there are any libraries included but are actually not needed, unless you have a tool to automatically check this.

2) the use of library effectively aggregates dependency on lower-level modules, but it also makes the dependency graph not accurate, which leads to overbuilding -- for instance, if you modify an object file in a library, every executable depending on the library has to be relinked but they may not all need that object.

I strongly believe dependency can be deduced from the source files (at least for c/c++). You may find my own build system interesting: http://code.google.com/p/qi-make/

Reply

nyork August 18, 2011 at 4:45 PM

@duff

It's the other way around: It is easy to generate a Makefile from BUILD files (in fact, the system worked this way at one time), but it is difficult (impossible?) to generate BUILDs file from a Makefile. This is because BUILD files provide dependencies between higher-level abstract entities (targets and packages), which would be difficult to infer from file based dependencies.

@Gilbert

Yes, (1) is potentially an issue, and is essentially a form of overbuilding as you mentioned in (2).

It is important to add that the build system and infrastructure is language agnostic. It works great for C++, Java, shell scripts, ... whatever. The only requirement is that each underlying tool executed (compiler, linker, etc) must obey certain rules (more on this in our next post). Fully declaring dependencies frees the build system from having to understand the semantics of each language, which it means we don't spend CPU time (and IO) reading and parsing source code on the workstation.

Our next post describes how we use the information from BUILD files to make the build extremely fast and efficient, which will also show why overbuilding is less of an issue that it may seem.

Reply

10587271141755126777 August 18, 2011 at 10:30 PM

To give a more concrete example of how BUILD rules are at a higher level than Make rules: notice that the BUILD file does not say whether a cc_library rule should create an archive library or a shared library. In fact, one might produce neither, yielding only object files, because the consuming actions might use gold's --start-lib/--end-lib feature (when performing a static link). So yes, any particular lowering to actions could be represented by a Makefile, but the BUILD language leaves us considerable freedom to change the details of that lowering. (Also, this file format doesn't assign meaning to tab characters.)

Overdeclaration of dependencies is indeed a problem with this system (although this is much better than permitting underspecified dependencies, which make incremental builds untrustworthy). Most module dependencies can be inferred from source code, but this is difficult and expensive for some languages, and it has turned out to be quite valuable to have a "cache" of this information in the form of BUILD files.

Reply

David August 19, 2011 at 1:55 AM

I see how BUILD rules are at a higher level than make rules. But I don't see how they allow greater parellelism. Could you give an example illustrating this?

Reply

nyork August 21, 2011 at 6:17 PM

@David

You guys are good. You're asking questions leading to our next post, which will cover this in finer detail. Without spoiling too much, let me provide a quick answer:

Fully exploiting parallelism in the build requires complete and correct dependency information. Without this, the build contains race conditions, where targets build out of order. In practice, many build related race conditions don't surface because the dependency graph traversal is predictable, or level of concurrency is insufficient, or some other happenstance means that the error is never encountered. For example, some GNUMake based builds only encounter errors with large values for "--jobs".

Higher level BUILD files do not magically fix this, but they do provide a better construct for dealing with the problem. BUILD targets list all inputs and dependences (outputs are implicit to the rule type and well defined). This means that we can more easily reason about exactly what each action should have access to, which means we can do things like sandbox action execution. The higher level rules in BUILD files provide a way to enforce better discipline in declaring inputs and dependencies, which indirectly enables greater parallelism.

Reply

SEA Limited April 18, 2013 at 12:16 AM

Your post is really well written. The examples which you mentioned for "How the Build System works" are perfect. I am looking forward to your next post :)

Investigative Engineering

Reply

Arun Kumar May 13, 2014 at 11:23 PM

Thanks for giving nice post

SEO Company Chennai

Reply

Global Act August 13, 2014 at 10:16 PM

Interesting Blog for Team Buildining Training

Reply

Yoga Fly To Light August 25, 2014 at 11:57 AM

It's really very informative that I wanted ever, thanks for this. If you want to build your rank fast watch this video

Reply

Enter your comment...

Comment as: Google Account

Publish Preview