Build as a Library

Sevki

Sun Dec 20 19:09:28 CES 2015

Abstract

Build as a library

I looked up my first commit

```
commit 2701e0c5126574a41e8abcbbe07ea81a74e51d7b
Author: Sevki Hasirci <s@sevki.org>
Date:   Thu Aug 13 20:57:34 2015 +0300

first commit
```

to see how long it has been since I started working on this and it has been a while, 4 months and some change to be vauge, 4 months is a long time to be working on something in the opensource world, but I have not been able to work on this a lot, since I was jumping trough a job interview after another, I ended up getting a job @ Spotify in Stockholm. During bootcamp and talking to fellow engineers I've told them about my new project and people have been interested, so I decided to start working on this more.

Objectives

Buids only purpose inlife is to provide you with build semantics and abstractions. That's it. So I call it *Build as a library*. The fact that it was written in go is a implementation detail, it very well could have been written in C, erlang, ruby and sure enough it was actually written in python and java(x2).

- Making a lightweight, 0 dependency build system so we can use it in plan9, harvey or freebsd run it on arm or powerpc or kubernetes.
- MapReduce'd distributed compilations and tests
- Better output

Before we go on I can't stress this enough, this is a **ALPHA** proof of concept software (even though the concept has been proven before).

First results and impressions

(excerpts from harvey mail list)

When I started testing this against the old harvey build system I got

```
CC=x86_64-elf-gcc build libs.json  17.75s user 7.66s system 90% cpu 28.087 total
```

compilation with build was

```
CC=x86_64-elf-gcc build //sys/src:libs  26.79s user 12.97s system 221% cpu 17.972 total
```

(I freaked out at first, how could my version be twice as slow as the old one I'm doing things in parallel, then I figured out I'm an idiot, and was reading the wrong column, mine is twice as fast as the old one. #winning)

Build introduces a lot of over head to the compilation process that wasn't there in the original build.go, It hashes all the files, reads the `stdout` and `stderr` of processes and does a bunch of other things. I'm pretty sure there can be some optimisations made to have major perf gains but I'm not too fussed about those.

So here is what it looks like when you run the old `build.go` and build

Video

and even if we just reduced the compilation time to half just with parallelising the compilation, that would be a good day in my book, but now minor edits take significantly less time, for instance,

Video

a minor comment change in anyfile and the original build.go will still compile in the same time

```
CC=x86_64-elf-gcc build libs.json  19.96s user 10.71s system 90% cpu 33.994 total
```

but build will compile it in a second and chnage

```
CC=x86_64-elf-gcc build //sys/src:libs  1.01s user 0.43s system 136% cpu 1.056 total
```

that's where the hashing and caching yield massive improovements.

Hashing Mechanics

before build builds a target, said target must provide a hash of its input, in the case of a c library build assumes that gcc or what ever compiler you are using is essentially a deterministic hash and for the given input lib, it should always compile the same binary, so in build target definition it tries to hash as many things as it can to make the hash we calculate as correct as possible, this includes, hashing all the files, hashing all the includes, all the parameters, copts and link opts as we can.

given that this is a build graph and every target is a node on that graph, we also produce hash for each node, that is the target and all its child dependencies hash, so if one of the dependencies is invalidated, every parent of that node will also be invalidated

so given the situation

```
a --> b --> c --> d  --> e
```

if 'e' is invalidated, all of it's parents will also be invalidated.

however it's not all roses, ATM build doesn't hash std libraries or system headers, which is not a problem for harvey since it doesn't use any of the std stuff, but that is just an implementation glitch that can easily be fixed by hashing those too. For instance when we first run build, cc library hashes the version of the `$CC`, that sort of technique can also be used to hash the std includes and libs.

so anything that requires a clean build is something build failed to include in its hash fucntion. I am also playing around with `libclang` that will abosulutely make sure that C/C++ stuff are properly hashed, using the actual hash function to produce the hash, as opposed to guessing it's validity but that's not a priority and it intoroduces a big dependency. That may just be for the server vesion.

Notes on go

while I was discussing this project the subject of lines of code popped up in the context of maintainability,

Project	files	blank	comment	code
bazel	2159	45111	103108	256772
buck	2836	55583	88187	281625
build	28	424	151	2307

while my code base is not as feature complete as its java counterparts, I don't think it will ever reach 100 times its current size.