Why are Android builds so slow?

source.dev

Expert-Led Innovation For
The Next Generation Of Devices

FOSDEM 2026

About me

Google

- Android OS engineer for nearly 10 years
- Android Runtime – Java, compiler optimizations
- Android Security – hypervisor, virtualization framework

source.dev

- tooling for device manufacturers who use Android

#1 complaint from engineers: **slow checkouts & builds**

- **Checkout: 15–20 minutes**
  - huge repository – 1,000+ Git projects managed by **repo** tool
  - AOSP mirror also quite slow + usage quota

- **Full build: 2 hours / $4.40**
  - huge codebase – 250,000+ build targets
  - faster hardware ⇒ faster build but higher cost

- **Incremental builds: depends**
  - indispensable during development
  - can be buggy due to incorrect dependencies

Test machine specs:

- c4-standard-32-lssd on GCP
- 32 vCPUs, 120 GB RAM, nVME SSD
- $2.20/hour

Test target:

- android16-qpr2-release
- aosp_cf_arm64_phone-bp4a-userdebug

Android + Bazel ??

source.dev

Expert-Led Innovation For
The Next Generation Of Devices

Soong as a step towards Bazel

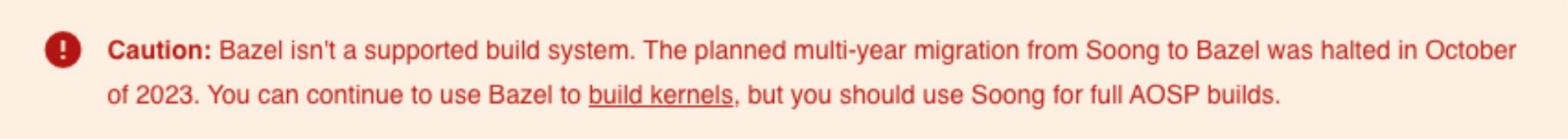
- Android wanted to migrate from Make to Bazel for a long time
  - hermetic/reproducible builds, build/test caching, scalability, …

- Original multi-year plan:
  - Step 1: develop Soong, using Bazel's file format but compat with Make
  - Step 2: gradually migrate Makefiles to new format
  - Step 3: start building parts of the OS with Bazel, phase out Soong/Make
  - Step 4: reap the benefits of Bazel and its associated infrastructure

- Culminated in Android 14 / early 2023
  - AOSP almost completely migrated to Soong, lower adoption downstream
  - Bazel builds appeared on ci.android.com

Android Builds – update on Bazel

And then…



Caution: Bazel isn't a supported build system. The planned multi-year migration from Soong to Bazel was halted in October of 2023. You can continue to use Bazel to build kernels, but you should use Soong for full AOSP builds.

So…

what now?

So what now?

- Soong != Bazel
  - hermeticity? sort of... reproducibility? sort of...
  - PATH sanitization, genrule sandbox, ...
  - but under the hood still compatible with Make ⇒ bypass

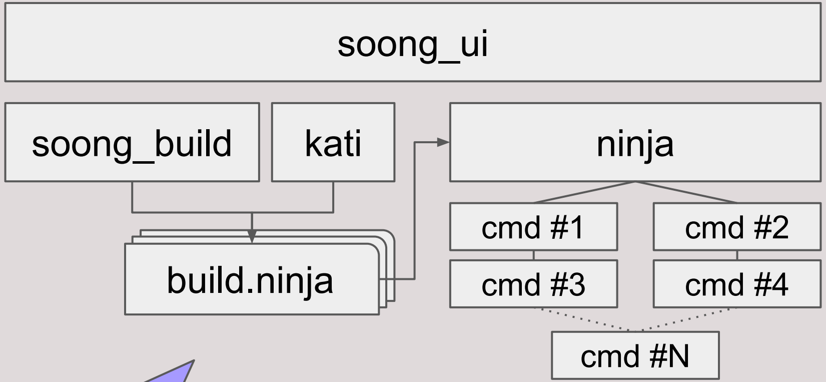
- Bazel builds are fast because of highly efficient caching
  - all inputs/outputs are declared ahead of time
  - easy to sandbox tasks, enforce dependencies, capture results
  - easy to offload to another machine (remote execution)

- So how much is possible without actually migrating to Bazel?

Android Builds – what you need to know

- Build files:
  - **Android.bp** (Blueprint) – similar to Bazel's Starlark, templates with logic in Go
  - **Android.mk** (Makefile) – legacy format, still used for device config, packaging

- Components:
  - **soong_ui** – top-level process, progress bar
  - **soong_build** – compiles Blueprint to Ninja
  - **kati** – compiles Makefile to Ninja
  - **ninja** – primary low-level runner



soong_ui

soong_build

kati

ninja

build.ninja

cmd #1

cmd #2

cmd #3

cmd #4

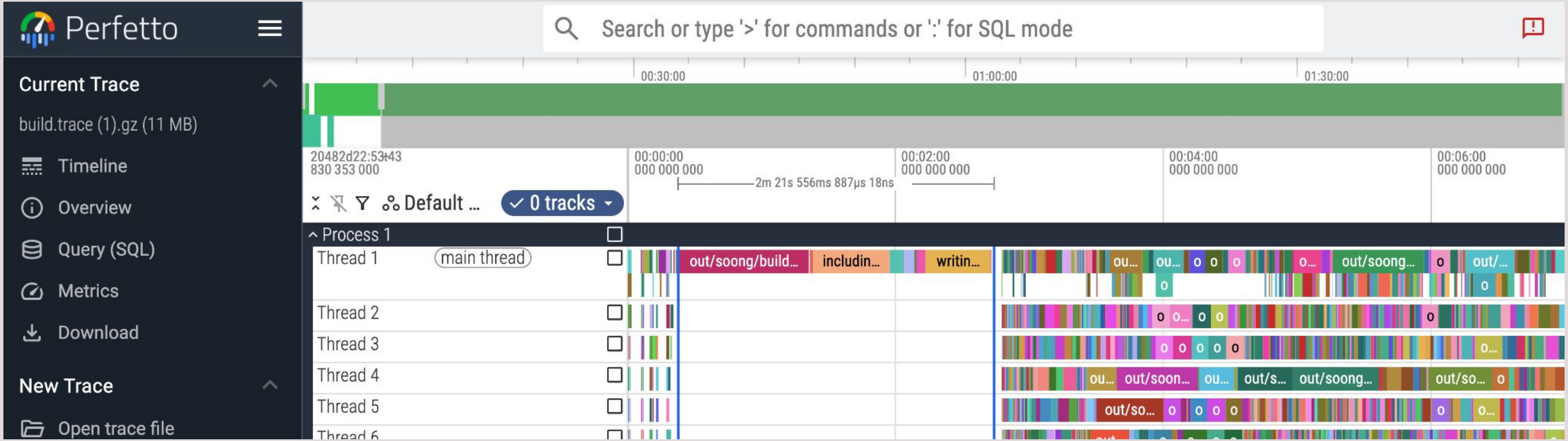
cmd #N

2-7 minutes
7GB of data

See Chris Simmonds' talk from EOSS 2023!!!

Android Builds – build traces

- Build trace extremely helpful in identifying long-running jobs
- Generated during a build at `out/build.trace.gz`
- Web UI: https://ui.perfetto.dev/



Android builds – rule of thumb for hardware

CPU

more is always better, 24/32 cores is a good baseline

more recent architecture is usually worth it

RAM

2GB for every logical core (64GB for 32 cores)

same size swap to handle occasional spikes

nVME

low latency is much more important than transfer speed

working directory has 4M files but median size is just 300 bytes!

`soong_build` scans 1M paths but only reads a fraction of those files

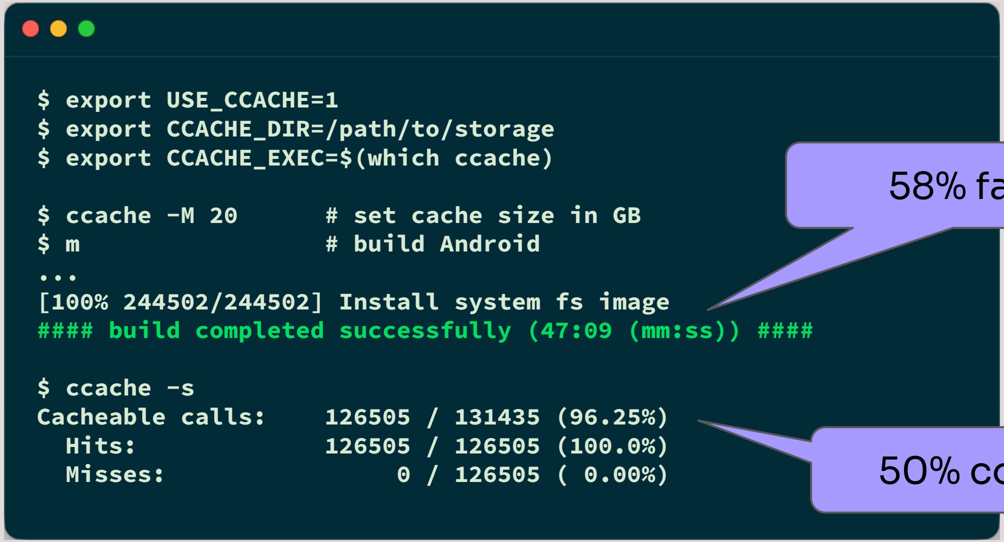
Caching with Soong

source.dev

Expert-Led Innovation For
The Next Generation Of Devices

Build caching – ccache

- Local cache, wrapper around C/C++ compilers (only)
- Supported by Android for a veeery long time
- No longer in `prebuilts/` but still works fine



```
$ export USE_CCACHE=1
$ export CCACHE_DIR=/path/to/storage
$ export CCACHE_EXEC=$(which ccache)

$ ccache -M 20      # set cache size in GB
$ m                 # build Android
...
[100% 244502/244502] Install system fs image
#### build completed successfully (47:09 (mm:ss)) ####

$ ccache -s
Cacheable calls:    126505 / 131435 (96.25%)
  Hits:             126505 / 126505 (100.0%)
  Misses:                0 / 126505 ( 0.00%)
```

58% faster

50% coverage

Build caching – reclient

reclient = retrofit of Bazel RBE API for other build systems

- compiler wrapper (just like ccache) which connects to RBE backend
- support for many compilers – clang, linker, javac, metalava, r8, d8, signapk, …

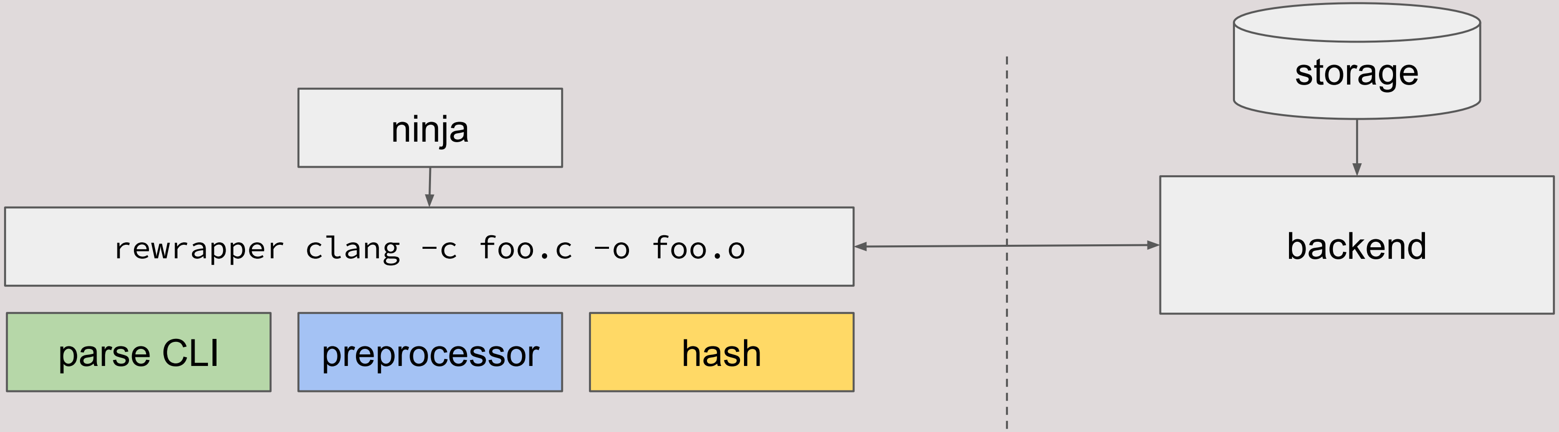
RBE services

- distributed build cache – exchange results of build steps
- remote execution – offload job to a remote server

Open-source backends

- Buildbarn, Buildfarm, BuildGrid (Apache 2.0)
- https://bazel.build/community/remote-execution-services

Build caching – reclient



Action

```
command: clang -c foo.c -o foo.o
env: [ PATH: “...”, PWD: “...”, ... ],
inputs: [
  (“foo.c”, “9f3c2a7”), (“clang”, “76fa3b1”),
  (“stdio.h”, “65e79f”), (“stdlib.h”, “143ee2f”)
]
```

ActionResult

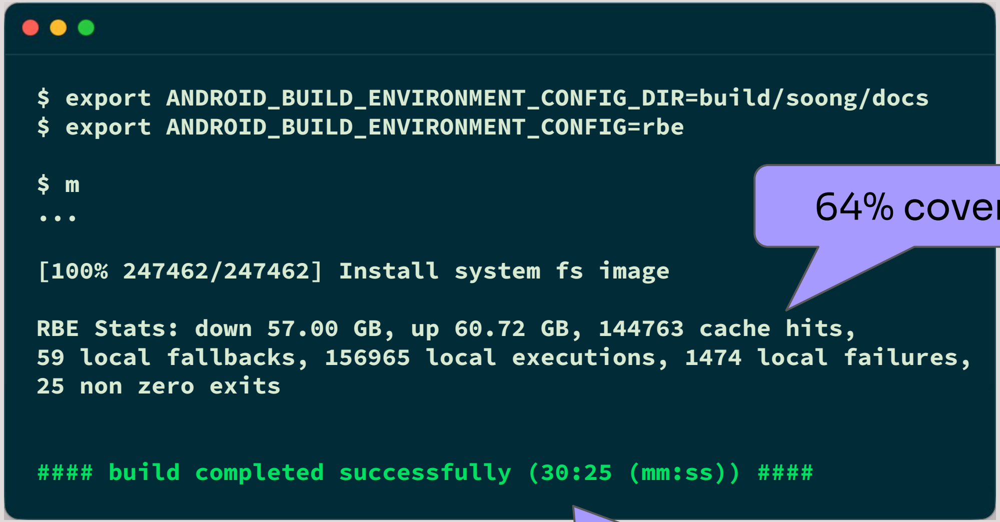
```
outputs: [
  (“foo.o”, “ac362aa7”),
  (“foo.o.d”, “99fa4be”)
]
```

Build caching – reclient

1. Configure and start an RBE backend
2. Create JSON config file
    - example in `build/soong/docs/rbe.json`
    - populate IP address, credentials
    - select toolchains
        - use cs.android.com to find the names
    - select exec strategies (on cache miss)
        - `remote_local_fallback` – remote execution
        - `local` - no remote execution
        - `racing` – both in parallel
3. Enable RBE with environment variables

```
{
 "env": {
  "USE_RBE": "1",
  "RBE_DIR": "prebuilts/remoteexecution-client/live",

  "RBE_service": "<ip_address>:<port>",
  "RBE_instance": "main",
  "RBE_service_no_security": "true",

  "RBE_D8": "1",
  "RBE_JAR": "1",
  "RBE_JAVAC": "1",
  "RBE_METALAVA": "1",
  "RBE_R8": "1",
  ...

  "RBE_CXX_EXEC_STRATEGY": "local",
  "RBE_D8_EXEC_STRATEGY": "local",
  "RBE_JAVAC_EXEC_STRATEGY": "local",
  "RBE_METALAVA_EXEC_STRATEGY": "local",
  "RBE_R8_EXEC_STRATEGY": "local",
  ...

  "RBE_log_dir": "/tmp",
  "RBE_output_dir": "/tmp",
  "RBE_proxy_log_dir": "/tmp"
 }
}
```

Build caching – reclient



```
$ export ANDROID_BUILD_ENVIRONMENT_CONFIG_DIR=build/soong/docs
$ export ANDROID_BUILD_ENVIRONMENT_CONFIG=rbe

$ m
...

[100% 247462/247462] Install system fs image

RBE Stats: down 57.00 GB, up 60.72 GB, 144763 cache hits,
59 local fallbacks, 156965 local executions, 1474 local failures,
25 non zero exits


#### build completed successfully (30:25 (mm:ss)) ####
```

64% coverage

73% faster

What next?

Better coverage, less pre-processing

- heterogeneous codebases (Android, Yocto, ...) need a generic solution
- requires a cheap mechanism to isolate and monitor each build command

Speeding up Ninja-file generation

- ideally the algorithm would get faster (anecdotally seems to have improved)
- extending caching to `soong_build/kati` would also help

Checkouts still a big problem, especially in CI

- prebuilts form 50% of a checkout but only one version of one architecture used
- Git VFS provides on-demand checkout of files that are actually needed

Thank You

Curious about source.dev?

We’re hiring!

jobs@source.dev