Accelerating Builds with Buck2

Neil Mitchell
Programmer, Meta

Meta

Buck2 is...

- A build system
- Developed and used by Meta
- Supports many languages (C++, Rust, Python, Go, OCaml, Erlang...)
- Designed for large mono repos
- Open source - buck2.build github.com/facebook/buck2
- 2x as fast as Buck1 😎

```
$ buck2 build //buck2:buck2
File changed: //buck2/app/buck2_core/src/cells/paths.rs
Network: Up: 2.4MiB 2.0MiB/s  Down: 568B
Command: build.                          Remaining: 138/27K. Cache hits: 72%. Time elapsed: 49.5s
--------------------------------------------------------------------------------
//buck2/app/buck2_query:buck2_query -- action (rustc metadata) [re_execute + 1]              2.4s
//buck2/app/buck2_events:buck2_events -- action (rustc link) [re_upload + 1]                 2.4s
//buck2/app/buck2_query:buck2_query -- action (rustc link) [re_upload + 1]                   2.4s
//buck2/app/buck2_test_api:buck2_test_api -- action (rustc link) [re_upload 0.2s + 1]        0.4s
//buck2/shed/more_futures:more_futures -- action (rustc link) [re_download 0.2s + 1]        0.4s
//buck2/app/buck2_test_api:buck2_test_api -- action (rustc metadata) [re_upload 0.2s + 1]    0.4s
//buck2/shed/more_futures:more_futures -- action (rustc metadata) [re_upload 0.2s + 1]       0.4s
```

//buck2:buck2 is a target, which *depends* on targets like
//buck2/app/buck2_events:buck2_events

Targets

```
# BUCK

rust_binary(
    name = "buck2",
    srcs =  ["bin/buck2.rs"],
    deps = [
        "//third-party/rust:anyhow",
        "//buck2/app/buck2_events:buck2_events",
        ...
    ],
)
```

Written in Starlark, aka deterministic simple Python

Core
Rust

Build graph

APIs

Starlark interpreter
- Profiling
- LSP/DAP
- Linter
- Typechecker

Console output

Logging/events

Performance!
- Parallelism
- Incrementality
- I/O
- Remote execution

API

Rules
Starlark

Rules from Meta are available,
but you can write your own

Libraries/binaries/tests

Supports many languages
- C++
- Python
- Rust
- Erlang
- OCaml
- Go
- Haskell
- …

Plus downloads, shell commands, aliases etc

Rules

Targets
Starlark

Written by the user

Specific to each project

Can use Starlark functions to abstract over common patterns

Faster!

- 2x as fast as Buck1 😎
- Waiting 10 minutes → 5 minutes ⏱️
- Engineers whose builds were sped up by Buck2 often produced meaningfully more code 💻

Performance 1 of 5: Abstraction

Good APIs mean you can optimize the core,
without rewriting the rules.

Starlark/Rust boundary is a strong abstraction.
Requires good API design, powerful APIs.
API should say what to do, but be insulated from how.

In Buck1, rules were written co-mingled with the core,
prevented optimisations.

Performance 2 of 5: Parallel + incremental

A single graph on which all computations live.

Computations are functions from keys to values,
which may access the values of other keys.

e.g. read a file, evaluate Starlark, run a command line

Those computations are run in parallel with dependency tracking.
Some computations (e.g. read file) can get invalidated externally.

Performance 3 of 5: Remote execution

Can run commands on an external server.

Reuse the Bazel Remote Execution API

★ CAS (Content Addressable Storage) maps hashes to files.
★ Execution server takes a hash of command line plus input files, runs it, producing output hashes.

More parallelism - can spawn 1000's of compiles.

More incrementality - execution server can also cache.

Performance 4 of 5: Virtual files

I/O is expensive!

Intermediate files don’t have to be downloaded if going remote.

Just use hashes in memory, download final result.

Integrates with Eden file system (from Meta) backed by version control. Hash a file without having it locally.

Maybe one day: use a virtual file system for the output too.

Performance 5 of 5: Avoid O(repo)

Big repo ⇒ O(repo) ≅ O(😴)

Checking modtime of every file in the repo is too slow.
Use inotify or Watchman (from Meta) to watch for changed files.
Watchman also knows about Eden.

Store graph reverse-dependencies for fast invalidation.

Never scan the entire repo - just the subset you use.

The good

- Powerful, fast, modern build system
- Actively developed
- Open source.
  - Diffs go upstream ~15 min
  - We accept PRs
  - Same as internal version (minus RE server)

The bad

- Changing build system is hard!
- New - only a few external users
- Some rules don’t work open source yet (Java, iOS)
- Integration with package managers a bit weak

Questions?

https://buck2.build

https://github.com/facebook/buck2

Meta