Buck2

Neil Mitchell and Chris Hopman
Programmers, 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 😎

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 tricks

1. Abstraction - rewrite the code without changing rules.
2. Single dependency graph for parallelism and incrementality.
3. Remote execution - with precomputed merkle trees and Blake3.
4. Virtual files
   a. Deferred materialisation - building without the bytes
   b. Virtual file system for input, Eden
5. Rust, so no GC
6. Nothing O(repo)

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

Fat Platforms

Fat platforms

Problem:

Same project built on Macs and on Linux, with naive solution:

- cache isn't shared, must build everything on both
- building on mac requires mac RE, which may be significantly more expensive than linux RE

Bad Solution: Run everything on linux by leveraging RE

- Resolves cache duplication + high mac RE cost
- If build host is mac, cannot use hybrid execution and build duration is slow

Q: Could we produce actions that can be run on either of Mac+Linux? Generally, on any platform?

Fat platforms

Solution: “Fat platforms” and fat tools

Basic Idea: A “fat tool” is a tool that can run on any of some set of platforms (for example Mac or Linux).

A “fat platform” is in a superset of platforms. Ex. a Mac+Linux fat platform.

Only fat tools are compatible with fat platforms.

Fat platforms

Two main approaches:

1. a naturally cross-platform binary (e.g. java, python)
2. a “fat binary” includes multiple platform-specific binaries, with one selected at runtime

For (2):

Can wrap a binary rule with a `fat_binary()` that performs a split transition on the underlying binary and then selects at runtime

Example: android ndk has prebuilt binaries for multiple platforms, easily wrapped into fat-platform supporting targets

Fat platforms

Recap: fat tools run on multiple platforms, “fat platforms” represent a platform that may be one of multiple possibilities (e.g. mac or linux)

Benefits: Ex. Devs can locally develop on macs, use hybrid execution, but still use linux RE. Builds from mac or linux share cache (as they both use the same fat exec platform).

Downsides: fat tools may be significantly larger than their non-fat counterparts

Slightly dynamic

Dynamic (monadic) dependencies are helpful. E.g. OCaml compilation, ThinLTO.

But, figuring out the blast radius of a change is super important for CI. Requires a static target graph.

Solution: We only add features that give dynamic action graph (dyamic_outputs) OR refine a graph (anonymous target).

Dynamic Outputs

dynamic_outputs()

Problem: Existing build apis require constructing static action graph, we cannot create actions or determine the inputs to actions based on the output of other actions.

Solution: dynamic_outputs():

- During analysis, artifacts can be bound as outputs to a dynamic_output (instead of a normal action)
- dynamic_output holds a starlark lambda
- when run, has access to the contents of its artifact inputs
- can declare new artifacts and actions and re-bind its own outputs to new actions

dynamic_output()

Example: filtering inputs

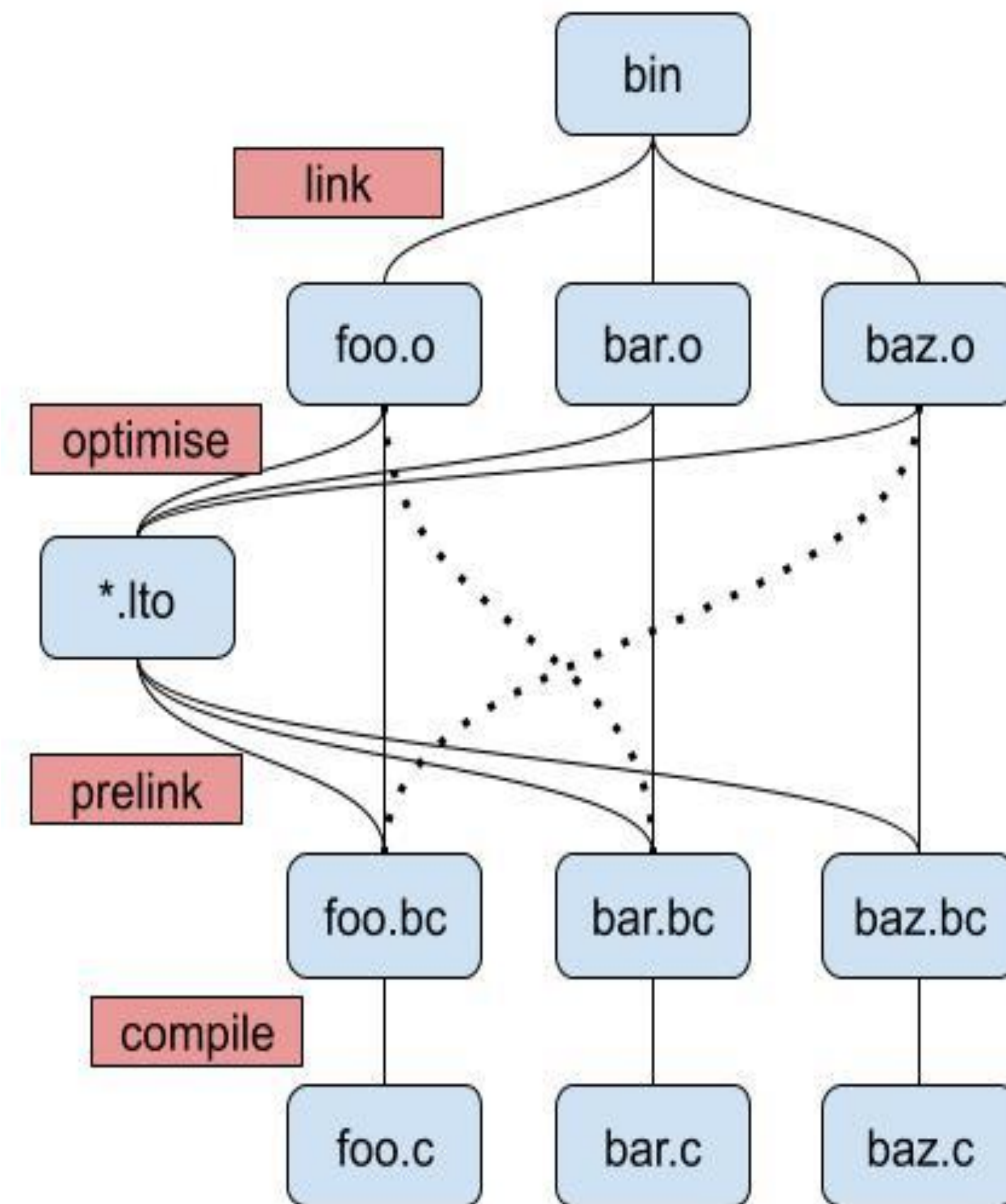
```
def _some_impl(ctx):
   inputs = ctx.attrs.includes
   primary_input = ctx.attrs.src
   output = ctx.actions.declare_output("output.o")
   required_inputs = ctx.actions.declare_output("inputs.x")
   ctx.actions.run([ctx.attrs.includes_filter, primary_input, required_inputs.as_output()])
   def f(ctx, artifacts, outputs):
     computed_inputs = artifacts[required_inputs].read_lines()
     filtered_inputs = filter_inputs(inputs, computed_inputs)
     ctx.actions.run([ctx.attrs.processor, primary_input, outputs[output]] +
   filtered_inputs)
   ctx.actions.dynamic_output(dynamic = [required_inputs], outputs = [output], f = f)
   return [DefaultInfo(default_outputs = [output])]
```

dynamic_output()

Real life examples:

- Distributed thin lto
    - bitcode optimizations dependencies are determined by the prelink

- Ocaml
    - Run ocamldeps to determine deps for compilation



Anonymous Targets

anon_target()

Problem: The right graph structure is not necessarily just defined by user written targets. You might want a shadow graph (aka aspects/overlays) or to share nodes.

Solution: anon_target():

- A target node defined by the hash of its attributes - no name.
- At analysis time you can depend on an anon_target

Used for Python native library sharing, Swift modules (require adding configuration to)

anon_target()

```
UpperInfo = provider(fields = ["message"])

def _impl_upper(ctx):
   return [UpperInfo(message = ctx.attrs.message.upper()]

upper = rule(
   attrs = {"message", attrs.string()},
   impl = _impl_upper
)

# Use an anonymous target
def impl(ctx):
   def k(providers):
       print(providers[UpperInfo].message)
       # These are the providers this target returns
       return [DefaultInfo()]
   return ctx.actions.anon_target(upper, {
       name: "my//:greeting",
       message: "Hello World",
   }).promise.map(k)
```

BXL

BXL

Problem:

Integrations (LSP, linters, binary/graph analysis, etc) are:

- very complex, needs lots of information encoded in build graph
- hard to define with existing build api
- don’t need to be consumable as build outputs

Solution:

BXL, a build extension language

BXL

Solution: BXL, a build extension language

- Starlark, with extensive APIs for interacting with buck's graph
- Leverages buck2 core's incremental caching engine
- API to define command line arguments

  `buck2 bxl my_custom_command.bxl -- --example-arg some.file --xxx-flag`
- Query and inspect unconfigured and configured graphs

  `ctx.uquery.eval()`, `ctx.cquery.eval()`, `ctx.cquery.kind()/filter()/deps()/etc`
- Access to analysis results

  `ctx.analysis(target).providers[DefaultInfo]`
- Declare new artifacts and actions

  `ctx.actions.declare_output()`, `ctx.actions.run()`

  outputs based on bxl script + cli args (instead of configured target)
- Integrated with dynamic features

  `ctx.actions.dynamic_outputs(), ctx.actions.anon_target()`

BXL

```
def _ebin_paths(ctx):
  target_universe = ctx.uquery().owner(ctx.cli_args.source)
  test_target = ctx.cquery().kind("erlang_test", ctx.cquery().owner(ctx.cli_args.source, target_universe))
  app_target = ctx.cquery().kind("erlang_app$", ctx.cquery().owner(ctx.cli_args.source, target_universe))

  target = test_target + app_target

  paths = []
  failed_targets = []
  for k, value in ctx.build(target).items():
    for _ in value.failures():
      failed_targets.append(str(k.raw_target()))
      break
    paths.extend(ctx.output.ensure_multiple(value.artifacts()))

  if failed_targets:
    fail("failed to build {} targets: {}".format(len(failed_targets), failed_targets))

  ctx.output.print(sep = "\n", *paths)

ebin_paths = bxl_main(
  impl = _ebin_paths,
  cli_args = {
    "source": cli_args.string(),
  },
)
```

BXL

Examples:

Erlang shell:

https://github.com/facebook/buck2/blob/main/prelude/erlang/shell/shell.bxl

Rust recursive check/clippy support:

https://github.com/facebook/buck2/blob/main/prelude/rust/rust-analyzer/check.bxl

Rust-analyzer support:

https://github.com/facebook/buck2/blob/main/prelude/rust/rust-analyzer/resolve_deps.bxl

Python sourcedb support:

https://github.com/facebook/buck2/tree/main/prelude/python/sourcedb

Incremental Actions

Incremental Actions

Problem:

binary-level actions are often O(repo)

Ex: linking, packaging, etc

Old Solution:

must design the tools such that the action can be broken up and expressed as smaller actions to an opinionated build system like buck

Incremental Actions

Problem:

binary-level actions are often O(repo)

- Ex: linking, packaging, etc

New Solution: Incremental Actions

- API that allows for lightweight incremental actions
- No need to encode the units of incrementality into build actions

Incremental Actions

How???

Two simple items:

1. no_outputs_cleanup = True

    An executing action can access the previous outputs

2. metadata_env_var/metadata_path

    buck provides hashes for all inputs

Expectation:

- The action writes as part of its output whatever it needs to compute its incremental update

Incremental Actions: Example

```
def _impl(ctx):
    json_srcs = ctx.actions.write_json("srcs.json", ctx.attrs.srcs, with_inputs = True)
    result = ctx.actions.declare_output("result", dir = True)
    state = ctx.actions.declare_output("incremental_state.json")
    command = cmd_args([ctx.attrs.script[RunInfo], "--output", result.as_output(),
"--incremental-state", state.as_output(), "--srcs", json_srcs])
    ctx.actions.run(command,  category = "x", metadata_env_var = "ACTION_METADATA",
metadata_path = "action_metadata.json",   no_outputs_cleanup = True)
    return [DefaultInfo(default_outputs=[result])]

sample_incremental = rule(
    impl = _impl,
    attrs = {
        "script": attrs.exec_dep(),
        "srcs": attrs.dict(attrs.source(), attrs.string()),

    }
)
```

Incremental Actions: Example

```
import json
import os
import shutil
import sys

prev_state = None
if os.path.exists(sys.argv[4]):
    with open(sys.argv[4]) as statefile:
        prev_state = json.load(statefile)

metadata = json.load(open(os.environ["ACTION_METADATA"]))
srcs = json.load(open(sys.argv[6]))

known_paths = {v["path"]: v["digest"] for v in metadata["digests"] if v["path"] in srcs.keys()}
unknown_paths = {v["path"]: v["digest"] for v in metadata["digests"] if v["path"] not in srcs.keys()}

# unknown paths includes srcs json and so the paths are part of the incremental key
key = [metadata["version"], sys.argv, unknown_paths]

if prev_state:
    if key != prev_state["key"]:
        prev_state = None

os.makedirs(sys.argv[2], exist_ok = True)
for path, h in known_paths.items():
    if prev_state and prev_state["srcs"][path] == h:
        print('skipping', path, file=sys.stderr)
        continue
    shutil.copyfile(path, os.path.join(sys.argv[2], srcs[path]))

with open(sys.argv[4], 'w') as statefile:
    json.dump({"key": key, "srcs": known_paths}, statefile)
```