Making build instrumentation boring with blight

William Woodruff

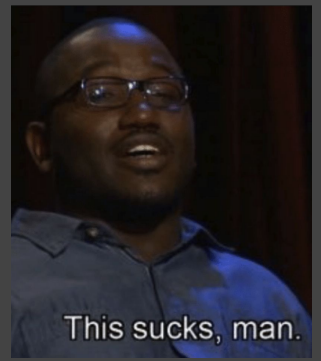
EH December 2020

Sometimes we need to instrument builds

- Caching: Need to divert to the compilation cache
  - `ccache`, `sccache`, others?
- Profiling: Which parts of the build are slowest?
  - Is duplicate work being done?
- Static analysis: need to track compilations
  - `compile_commands.json` (CMake w/ Make, Bear)
- Injected instrumentation: need to inject flags/code

Correct build instrumentation is a PITA

- **Diversity of build & metabuild systems**
  - CMake, Make, Bazel, your coworker's `bash` scripts, …
- **Diversity of compiler frontends**
  - GCC, Clang, ICC, MSVC, some vendor you're overpaying
  - Complicated CLIs with a lot of unusual argument parsing
- **Diversity of important compiler-adjacent tools**
  - Preprocessor, linker, archiver, assembler, …



blight: high-fidelity build instrumentation

- (meta-)build agnostic: doesn't care how it's run
- Minimally invasive: no `LD_PRELOAD` chicanery
- Support for `CC`, `CXX`, `CPP`, `LD`, `AS`, `AR`
  - Hi-fi models of each's behavior: inputs, outputs, opt level, ...
- Configurable actions that can be run on any/all tools
  - Built in: profiling, recording (cf. Bear), flag injection, flag removal

Actually using it

```
$ pip install blight
$ eval $(blight-env --guess-wrapped)
$ export BLIGHT_ACTIONS="Record"
$ export BLIGHT_ACTION_RECORD="output=/tmp/demo.jsonl"
$ make
```

- **Also supported:**
  - builds that hardcode **`gcc`**, etc.
  - multiple actions per run, e.g `BLIGHT_ACTIONS=”IgnoreWerror:Record”`

Demo: recording a deeply nested build

Links

- GitHub: trailofbits/blight
- Blog post: High-fidelity build instrumentation with blight
- Docs: trailofbits.github.io/blight
- PyPI: pypi.org/project/blight