Rattle

Simpler builds for smaller use cases

Neil Mitchell
https://ndmitchell.com/

Build two C files and link

```
$ cat make.sh
gcc -c main.c
gcc -c util.c
gcc -o main.exe main.o util.o
```

Shell script

- Simple to write
- Full control over commands

Build system

- More complex
- Must specify dependencies
  - E.g. header files, toolchain

But you gain:

- Parallelism
- Incrementality

Introducing Rattle

```
$ rattle make.sh
```

Gives you parallelism, incrementality, cloud builds.

https://github.com/ndmitchell/rattle

Build Systems with Perfect Dependencies,
Sarah Spall, Neil Mitchell and Sam Tobin-Hochstadt
OOPSLA 2020

Build Scripts with Perfect Dependencies

SARAH SPALL, Indiana University, USA
NEIL MITCHELL, Facebook, UK
SAM TOBIN-HOCHSTADT, Indiana University, USA

Build scripts for most build systems describe the actions to run, and the dependencies between those actions—but often build scripts get those dependencies wrong. Most build scripts have both *too few dependencies* (leading to incorrect build outputs) and *too many dependencies* (leading to excessive rebuilds and reduced parallelism). Any programmer who has wondered why a small change led to excess compilation, or who resorted to a "clean" step, has suffered the ill effects of incorrect dependency specification. We outline a build system where dependencies are *not specified*, but instead *captured by tracing execution*. The consequence is that dependencies are always correct by construction and build scripts are easier to write. The simplest implementation of our approach would lose parallelism, but we are able to recover parallelism using speculation.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Additional Key Words and Phrases: build systems, functional programming

ACM Reference Format:
Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. 2020. Build Scripts with Perfect Dependencies. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 169 (November 2020), 28 pages. https://doi.org/10.1145/3428237

1 INTRODUCTION

Every non-trivial piece of software includes a "build system", describing how to set up the system from source code. Build scripts [Mokhov et al. 2018] describe *commands to run* and *dependencies to respect*. For example, using the Make build system [Feldman 1979], a build script might look like:

```
main.o: main.c
    gcc -c main.c
util.o: util.c
    gcc -c util.c
main.exe: main.o util.o
    gcc -o main.exe main.o util.o
```

This script contains three rules. Looking at the first rule, it says `main.o` depends on `main.c`, and is produced by running `gcc -c main.c`. What if we copy the commands into a shell script? We get:

```
gcc -c main.c
gcc -c util.c
gcc -o main.exe main.o util.o
```

That's shorter, simpler and easier to follow. Instead of declaring the outputs and dependencies of each command, we've merely given one valid ordering of the commands (we could equally have put `gcc -c util.c` first). This simpler specification has additional benefits. First, we've fixed

Authors' addresses: Sarah Spall, Indiana University, USA, sjspall@iu.edu; Neil Mitchell, Facebook, UK, ndmitchell@gmail.com; Sam Tobin-Hochstadt, Indiana University, USA, samth@cs.indiana.edu.

2475-1421/2020/11-ART169
https://doi.org/10.1145/3428237

How to get incrementality?

- The script runs a series of commands
    - The future commands can depend on the result of previous commands (dynamic dependencies)
- For each command, Rattle records the inputs/outputs using fsatrace
    - Syscall hooking, LD_LIBRARY_PRELOAD, Windows hooks
- Next time it encounters that command, if no inputs have changed, the outputs are reused
    - Assumes commands are deterministic

Fabricate was one of the first build systems to do this trick.

How to get cloud builds?

- Whenever we run a command, we store the inputs/outputs in a cloud cache
- Before running a command, if any command matches, download the outputs

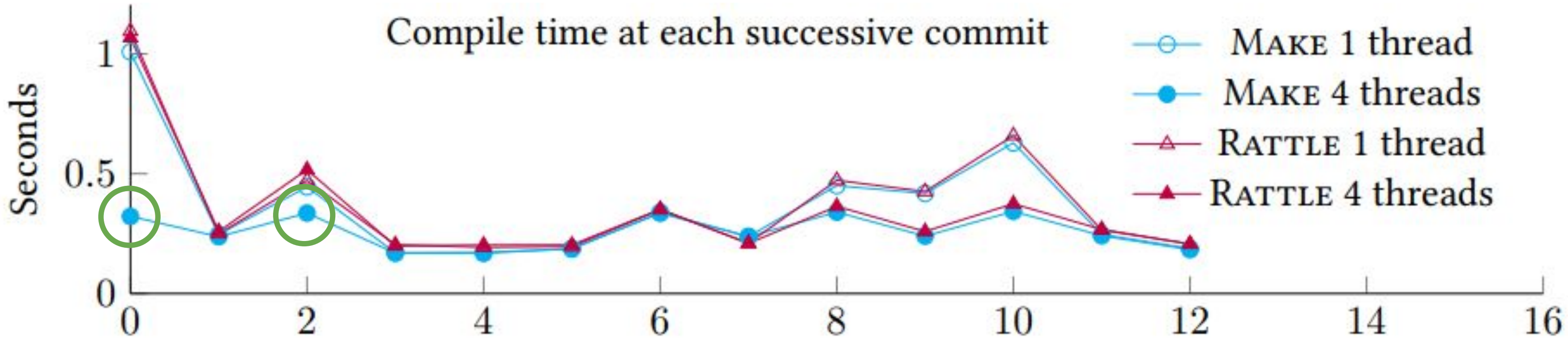
Not quite as simple as it seems… Some inputs (e.g. C files) may change which other inputs are required (e.g. header files). But (at worst) just scan for a match.

How to get parallelism?

The tricky one!

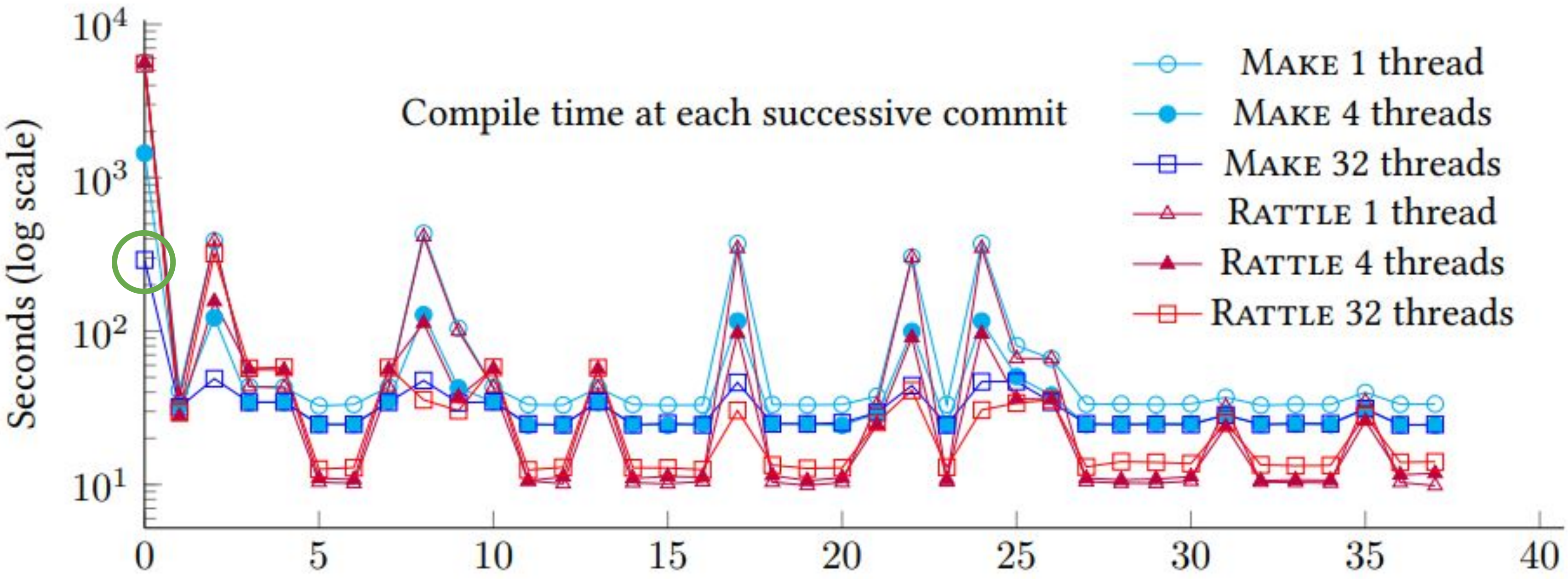
- Guess what commands will come next. Run them. See if you were right.
- Speculation - think of the CPU speculating on instructions
  - And remember how that has turned out - lots of tricky details
- For speculation to be valid, we need to know certain properties about commands
  - E.g. doesn’t read a file that hasn’t yet been written
  - The paper introduced “hazards” and proves the necessary properties, Rattle checks them
  - If hazards trip you up, rerun (speed hit)

Does it work? FSATrace



Same time as Make, despite not having the commit info

Does it work? Node.js



Faster than make, because dependencies are precise

Why “small” use cases?

- Immature technology (technology preview really)
- Must give a single linearisable trace
  - Doing that *compositionally* at scale often requires dependencies

Rattle makes it easy to do a simple build system.

Sweet spot might be small open-source multi-language projects?