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

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.

some potential bugs – these commands depend on the undeclared dependency gcc, and whatever header files are used by main.c and util.c. These bugs could be fixed in the Makefile by adding a dependency on the version of gcc used, as well as by listing every header file transitively included by main.c and util.c Furthermore, as the files main.c and util.c evolve, and their dependencies change (by changing the #include directives), the shell script remains correct, while the MAKE script *must* be kept consistent or builds will become incorrect.

1.1 Why dependencies?

Given the manifest simplicity of the shell script, why write a Makefile? Build systems such as MAKE have two primary advantages, both provided by dependency specification: **incrementality** and **parallelism**. MAKE is able to re-run only the commands needed when a subset of the files change, saving significant work when only util.c changes. MAKE can also run multiple commands in parallel when neither depends on the other, such as the two invocations of gcc -c. With a shell script, these are both challenging research problems in incremental computing and automatic parallelization, respectively, and unlikely to be solvable for arbitrary programs such as gcc.

1.2 Builds without dependencies

In this paper we show how to take the above shell script and gain many of the benefits of a MAKE build (§2). Firstly, we can skip those commands whose dependencies haven't changed by *tracing* which files they read and write (§2.4) and keeping a history of such traces. Secondly, we can run some commands in parallel, using *speculation* to guess which future commands won't interfere with things already running (§2.8). The key to speculation is a robust model of what "interfering" means – we call a problematic interference a *hazard*, which we define in §2.6.

We have implemented these techniques in a build system called RATTLE[1], introduced in §3, which embeds commands in a Haskell script. A key part of the implementation is the ability to trace commands, whose limitations we describe in §3.5. We show that our design decisions produce correct builds by formalizing hazards and demonstrating important properties about the safety of speculation in §4. To evaluate our claims, and properly understand the subtleties of our design, we converted existing MAKE scripts into RATTLE scripts, and discuss the performance characteristics and MAKE script issues uncovered in §5. We also implement two small but non-trivial builds from scratch in RATTLE and report on the lessons learned. Our design can be considered a successor to the MEMOIZE build system [McCloskey 2008], and we compare RATTLE with it and other related work in §6. Finally, in §7 we conclude and describe future work.

2 BUILD SCRIPTS FROM COMMANDS

Our goal is to design a build system where a build script is simply a list of commands. In this section we develop our design, starting with the simplest system that just executes all the commands in order, and ending up with the benefits of a conventional build system.

2.1 Executing commands

Given a build script as a list of commands, like in §1, the simplest execution model is to run each command sequentially in the order they were given. Importantly, we require the list of commands is ordered, such that any dependencies are produced before they are used. We consider this sequential execution the reference semantics, and as we develop our design further, require that any optimised/cached implementation gives the same results.

---

[1]https://github.com/ndmitchell/rattle

2.2 Value-dependent commands

While a static list of commands is sufficient for simple builds, it is limited in its expressive power. Taking the build script from §1, the user might really want to compile and link *all* .c files – not just those explicitly listed by the script. A more powerful script, along with its translation to working RATTLE code, is shown in figure 1.

```
FILES=$(ls *.c)                 main = rattle $ do
for FILE in $FILES; do            cs <- liftIO $ getDirectoryFiles "." [root </> "*.c"]
    gcc -c $FILE                  forP cs $ \c -> cmd "gcc -c" c
done                              let toO x = takeBaseName x <.> "o"
gcc -o main.exe ${FILES%.c}.o     cmd "gcc -o main.exe" (map toO cs)
```

Fig. 1. Value-dependent build scripts, in Shell and Rattle

This script now has a curious mixture of commands (ls, gcc), control logic (for) and simple manipulation (changing file extension[2]). Importantly, there is *no fixed list of commands* – the future commands are determined based on the results of previous commands. Concretely, in this example, the result of ls changes which gcc commands are executed. The transition from a fixed list of commands to a dynamic list matches the Applicative vs Monadic distinction of Mokhov et al. [2018, §3.5].

There are three approaches to modelling a dynamic list of commands:

(1) We could consider the commands as a stream given to the build system one by one as they are available. The build system has no knowledge of which commands are coming next or how they were created. In this model, a build script supplies a stream of commands, with the invariant that dependencies are produced before they are used, but provides no further information. The main downside is that it becomes impossible to perform any analysis that might guide optimisation.

(2) We could expose the full logic of the script to the build system, giving a complete understanding of what commands are computed from previous output, and how that computation is structured. The main downside is that the logic between commands would need to be specified in some constrained domain-specific language (DSL) in order to take advantage of that information. Limiting build scripts to a specific DSL complicates writing such scripts.

(3) It would be possible to have a hybrid approach, where dependencies between commands are specified, but the computation is not. Such an approach still complicates specification (some kind of dependency specification is required), but would allow some analysis to be performed.

In order to retain the desired simplicity of shell scripts, we have chosen the first option, modelling a build script as a sequence of commands given to the build system. Future commands may depend on the results of previous commands in ways that are not visible to the build system. The commands are produced with "cheap" functions such as control logic and simple manipulations, for example, using a for loop to build a list of object files. We consider the cheap commands to be fixed overhead, run on every build, and not cached or parallelised in any way. If any of these cheap manipulations becomes expensive, they can be replaced by a command, which will then be visible to the build system. The simple list of commands from §1 is a degenerate case of no interesting logic between commands.

---

[2]We take some liberties with shell scripts around replacing extensions, so as not to obscure the main focus.

An important consequence of the control logic not being visible to the build system is that the build system has no prior knowledge of which commands are coming next, or if they have changed since it last executed them. As a result, even when the build is a simple static script such as from §1, when it is manually edited, the build will execute correctly. The build system is unaware if you edited the script, or if the commands were conditional on something that it cannot observe. Therefore, this model solves the problem of self-tracking from Mokhov et al. [2018, §6.5].

2.3 Dependency tracing

For the rest of this section we assume the existence of *dependency tracing* which can tell us which files a command accesses. Concretely, we can run a command in a special mode such that when the command completes (and not before) we can determine which files it read and wrote; these files are considered to be the command's inputs and outputs respectively. We cannot determine at which point during the execution these files were accessed, nor which order they were accessed in. We cannot prevent or otherwise redirect an in-progress access. We discuss the implementation of dependency tracing, and the reasons behind the (frustrating!) limitations, in §3.5.

2.4 Skipping unnecessary commands

When running a command, we can use dependency tracing to capture the files which that command reads and writes, and then after the command completes, record the cryptographic hashes of the contents of those files. If the same command is ever run again, and the inputs and outputs haven't changed (have the same hashes), it can be skipped. This approach is the key idea behind both Memoize [McCloskey 2008] and Fabricate [Hoyt et al. 2009]. However, this technique makes the assumption that commands are pure functions from their inputs to their outputs, meaning if a command's input files are the same as last time it executed, it will write the same values to the same set of files. Below are four ways that assumption can be violated, along with ways to work around it.

Non-deterministic commands Many commands are non-deterministic – e.g. the output of ghc object files contains unpredictable values within it (a consequence of the technique described by Augustsson et al. [1994]). We assume that where such non-determinism exists, any possible output is equally valid.

Incorporating external information Some commands incorporate system information such as a timestamp, so a cached value will be based on the first time the command was run, not the current time. For compilations that embed the timestamp in metadata, the first timestamp is probably fine. For commands that really want the current time, that step can be lifted into the control logic (as per §2.2) so it will run each time the build runs. Similarly, commands that require unique information, e.g. a GUID or random number, can be moved into control logic and always run.

Reading and writing the same file If a command both reads and writes the same file, and the information written is fundamentally influenced by the file that was read, then the command never reaches a stable state. As an example, echo x >> foo.txt will append the character x every time the command is run. Equally, there are also commands that read the existing file to avoid rewriting a file that hasn't changed (e.g. ghc generating a .hi file) and commands that can cheaply update an existing file in some circumstances (the Microsoft C++ linker in incremental mode). We make the assumption that if a command both reads and writes to a file, that the read does not meaningfully influence the write, otherwise it is not really suitable as part of a build system because the command never reaches a stable state and will re-run every time the build is run.

The most commonly used tool which violates these assumptions is Latex, where indeed, it is possible for every run of the Latex compiler to produce a new result. One common approach to

Latex is to run the compiler a fixed number of times (e.g. three), and a single command that ran Latex repeatedly would then obey our rules. There may be better approaches to handling tools such as Latex, but we leave that for future research.

Simultaneous modification If a command reads a file, but before the command completes something else modifies the file (e.g. a human or untracked control logic), then the final hash will not match what the command saw. It is possible to detect such problems with reads by ensuring that the modification time after computing the hash is before the command was started. For simultaneous writes the problem is much harder, so we require that all files produced by the build script are not simultaneously written to.

In general we assume all commands given to the build system are well behaved and meet the above assumptions.

2.5 Cloud builds

We can skip execution of a command if all the files accessed have the same hashes as any previous execution (§2.4). However, if *only the files read* match a previous execution, and the files that were written have been stored away, those stored files can be copied over as outputs *without* rerunning the command. If that storage is on a server, multiple users can share the results of one compilation, resulting in cloud build functionality. While this approach works well in theory, there are some problems in practice.

Machine-specific outputs Sometimes a generated output will only be applicable to the machine on which it was generated – for example if a compiler auto-detects the precise chipset (e.g. presence of AVX2 instructions) or hardcodes machine specific details (e.g. the username). Such information can often be lifted into the command line, e.g. by moving chipset detection into the control logic and passing it explicitly to the compiler. Alternatively, such commands can be explicitly tagged as not being suitable for cloud builds.

Relative build directories Often the current directory, or user's profile directory, will be accessed by commands. These directories change if a user has two working directories, or if they use different machines. We can solve this problem by having a substitution table, replacing values such as the users home directory with $HOME. If not rectified, this issue reduces the reusability of cloud results, but is otherwise not harmful.

Non-deterministic builds If a command has non-deterministic output, then every time it runs it may generate a different result. Anything that transitively depends on that output is likely to also vary on each run. If the user temporarily disconnects from the shared storage, and runs a non-deterministic command, even if they subsequently reconnect, it is likely anything transitively depending on that command will never match again until after a clean rebuild. There are designs to solve this problem (e.g. the modification comparison mechanism from Erdweg et al. [2015]), but the issue only reduces the effectiveness of the cloud cache, and usually occurs with intermittent network access, so can often be ignored.

2.6 Build consistency

As stated in Mokhov et al. [2018, §3.6], a build is correct provided that:

> If we recompute the value of the key (...), we should get exactly the same value as we see in the final store.

In our build system, a key is a filename, and a value are the files contents.

Specifying this requirement in terms more applicable to our design, it means that after a build completes, an immediate subsequent rebuild should have no effect because all commands are

skipped (assuming the commands given to the build system are the same). However, there are sequences of commands, where each command meets our assumptions separately (as per §2.4), but the combination is problematic:

```
echo 1 > foo.txt
echo 2 > foo.txt
```

This program writes 1 to foo.txt, then writes 2. If the commands are re-executed then the first command reruns because its output changed, and after the first command reruns, now the second commands output has changed. More generally, if a build writes different values to the same file multiple times, it is not correct by the above definition, because on a rebuild both commands would re-run. But even without writing to the same file twice, it is possible to have an incorrect build:

```
sha1sum foo.txt > bar.txt
sha1sum bar.txt > foo.txt
```

Here sha1sum takes the SHA1 hash of a file, first taking the SHA1 of foo.txt and storing it in bar.txt, then taking the SHA1 of bar.txt and storing it in foo.txt. The problem is that the script first reads from foo.txt on the first line, then writes to foo.txt on the second line, meaning that when the script is rerun the read of foo.txt will have to be repeated as its value has changed.

Writing to a file after it has already been either read or written is the only circumstance in which a build, where every individual command is well-formed (as per §2.4), is incorrect. We define such a build as *hazardous* using the following rules:

Read then write If one command reads from a file, and a later command writes to that file, on a future build, the first command will have to be rerun because its input has changed. This behaviour is defined as a *read-write hazard*. We assume the build author ordered the commands correctly, if not the author can edit the build script.

Write then write If two commands both write to the same file, on a future build, the first will be rerun (its output has changed), which is likely to then cause the second to be rerun. This behaviour is defined as a *write-write hazard*.

Using tracing we can detect hazards and raise errors if they occur, detecting that a build system is incorrect before unnecessary rebuilds occur. We prove that a build system with deterministic control logic, given the same input the control logic will produce the same output, and with no hazards always results in no rebuilds in §4.2. The presence of hazards in a build does not guarantee that a rebuild will always occur, for example if the write of a file after it is read does not change the file's value. But, such a build system is malformed by our definition and if the write definitely can't change the output, then it should not be present.

2.7 Explicit Parallelism

A build script can use explicit parallelism by giving further commands to the build system before previous commands have completed. For example, the script in §2.2 has a for loop where the inner commands are independent and could all be given to the build system simultaneously. Such a build system with explicit parallelism must still obey the invariant that the inputs of a command must have been generated before the command is given, requiring some kind of two-way feedback that an enqueued command has completed.

Interestingly, given complete *dependency* information (e.g. as available to Make) it is possible to infer complete *parallelism* information. However, the difficulty of specifying complete dependency information is the attraction of a tracing based approach to build systems.

2.8 Implicit Parallelism (Speculation)

While explicit parallelism is useful, it imposes a burden on the build script author. Alternatively we can use implicit parallelism, where some commands are executed speculatively, before they are required by the build script, in the hope that they will be executed by the script in the future and thus skipped by the build system. Importantly, such speculation can be shown to be safe by tracking hazards, provided we introduce a new hazard *speculative-write-read*, corresponding to a speculative command writing to a file that is later read by a command required by the build script (defined precisely in §4.1).

Given a script with no hazards when executed sequentially, we show in §4.2: 1) that any ordering of those commands that also has no hazards will result in an equivalent output, see Claim 3; 2) that any parallel or interleaved execution without hazards will also be equivalent, see Claim 4; and 3) if any additional commands are run that don't cause hazards, they can be shown to not change the results the normal build produces, see Claim 5. Finally, we prove that if a series of commands contains hazards, so will any execution that includes those required commands, see Claim 6.

As a consequence, if we can predict what commands the build script will execute next, and predict that their execution will not cause hazards, it may be worth speculatively executing them. Effective speculation requires us to predict the following pieces of data.

Future commands The benefit of speculatively executing commands is that they will subsequently be skipped, which only happens if the speculative command indeed occurs in the build script. The simplest way to predict future commands is to assume that they will be the same as they were last time. Given more information about the build, e.g. all the control logic as per §2.2 choice 2, it would be possible to use static analysis to refine the prediction.

Absence of hazards If a hazard occurs the build is no longer correct, and remediation must be taken (e.g. rerunning the build without speculation, see §4.3). Therefore, performance can be significantly diminished if a speculative command leads to a hazard. Given knowledge of the currently running commands, and the files all commands accessed in the last run, it is possible to predict whether a hazard will occur if the access patterns do not change. If tracing made it possible to abort runs that performed hazardous accesses then speculation could be unwound without restarting, but such implementations are difficult (see §3.5).

Recovering from Hazards caused by speculation If a build using speculative execution causes a hazard, it is possible that the hazard is entirely an artefact of speculation. There are a few actions the build system could take to recover and these are discussed in section §4.3.

3 IMPLEMENTING RATTLE

We have implemented the design from §2 in a build system called Rattle. We use Haskell as the host language and to write the control logic. Copying the design for Shake [Mitchell 2012], a Rattle build script is a Haskell program that uses the Rattle library.

3.1 A Rattle example

A complete Rattle script that compiles all .c files like §2.2 is given in Figure 2, with the key API functions in Figure 3. Looking at the example, we see:

- A Rattle script is a Haskell program. It makes use of ordinary Haskell imports, and importantly uses Development.Rattle, offering the API from Figure 3.
- The rattle function takes a value in the Run monad and executes it in IO. The Run type is the IO monad, enriched with a ReaderT [Jones 1995] containing a reference to shared mutable state (e.g. which commands are in flight, where to store metadata, location of shared storage).

```
import Development.Rattle
import System.FilePattern
import System.FilePath

main = rattle $ do
    cs <- liftIO $ getDirectoryFiles "." [root </> "*.c"]
    forP cs $ \c ->
        cmd "gcc -c" c
    let toO x = takeBaseName x <.> "o"
    cmd "gcc -o main.exe" (map toO cs)
```

Fig. 2. A Haskell/Rattle version of the script from §2.2

```
-- The Run monad                          -- Reading/writing files
data Run a = ...                          cmdReadFile :: FilePath -> Run String
rattle :: Run a -> IO a                   cmdWriteFile :: FilePath -> String -> Run ()

-- Running commands                       -- Parallelism
data CmdOption = Cwd FilePath | ...       forP :: [a] -> (a -> Run b) -> Run [b]
cmd :: CmdArguments args => args
```

Fig. 3. The Rattle API

- All the control logic is in Haskell and can use external libraries – e.g. System.FilePath for manipulating FilePath values and System.FilePattern for directory listing. Taking the example of replacing the extension from .c to .o, we are able to abstract out this pattern as toO and reuse it later. Arbitrary Haskell IO can be embedded in the script using liftIO. All of the Haskell code (including the IO) is considered control logic and will be repeated in every execution.
- Commands are given to the build system part of Rattle using cmd. We have implemented cmd as a variadic function [Kiselyov 2015] which takes a command as a series of String (a series of space-separated arguments), [String] (a list of arguments) and CmdOption (command execution modifiers, e.g. to change the current directory), returning a value of type Run (). The function cmd only returns once the command has finished executing (whether that is by actual execution, skipping, or fetching from external storage).
- We have used forP in the example, as opposed to forM, which causes the commands to be given to Rattle in parallel (§2.7). We could have equally used forM and relied on speculation for parallelism (§2.8).

Looking at the functions from Figure 3, there are two functions this example does not use. The cmdWriteFile and cmdReadFile functions are used to perform a read/write of the file system through Haskell code, causing hazards to arise if necessary. Apart from these functions, it is assumed that all Haskell control code only reads and writes files which are not written to by any commands.

For further examples of Rattle programs see §5.9.

3.2 Alternative Rattle wrappers

Given the above API, combined with the choice to treat the control logic as opaque, it is possible to write wrappers that expose Rattle in new ways. For example, to run a series of commands from a file, we can write:

```
main = rattle $ do
    [x] <- liftIO getArgs
    src <- readFile x
    forM_ (lines src) cmd
```

Here we take a command line argument, read the file it points to, then run each command sequentially using forM_. We use this script for our evaluation in §5.

An alternative API could be provided by opening up a socket, and allowing a RATTLE server to take command invocations through that socket. Such an API would allow writing RATTLE scripts in other languages, making use of the existing RATTLE implementation. While such an implementation should be easy, we have not yet implemented it.

3.3 Specific design choices and consequences

Relative to the reference design in §2 we have made a few specific design choices, mostly in the name of implementation simplicity:

- All of our predictions (see §2.8) only look at the very last run. This approach is simple, and in practice, seems to be sufficient – most build scripts are run on very similar inputs most of the time.
- We run a command speculatively if 1) it hasn't been run so far this build; 2) was required in the last run; and 3) doesn't cause a hazard relative to both the completed commands and predicted file accesses of the currently running commands. Importantly, if we are currently running a command we have never run before, we don't speculate anything – the build system has changed in an unknown way so we take the cautious approach.
- We never attempt to recover after speculation (see §4.3), simply aborting the build and restarting without any speculation.
- We treat command lines as black boxes, never examining them to predict their outputs. For many programs a simple analysis (e.g. looking for -o flags) might be predictive.
- RATTLE can go wrong if a speculated command writes to an input file, as per §4.3. This problem hasn't occurred in practice, but dividing files into inputs and outputs would be perfectly reasonable. Typically the inputs are either checked into version control or outside the project directory, so that information is readily available.
- It is important that traces (as stored for §2.4) are only recorded to disk when we can be sure they were not affected by any hazards (see §4.3). That determination requires waiting for all commands which ran at the same time as the command in question to have completed.
- We model queries for information about a file (e.g. existence or modification time) as a read for tracing purposes, thus depending on the contents of the file. For queries about the existence of a file, we rerun if the file contents changes, which may be significantly more frequent than when the file is created or deleted. For queries about modification time, we don't rerun if the modification time changes but the file contents don't, potentially not changing when we should. In practice, most modification time queries are to implement rebuilding logic, so can be safely ignored if the file contents haven't changed.

3.4 Storage and cloud storage

Our RATTLE implementation stores all information required between executions to disk. To provide reuse between multiple different users, we can store these artefacts on a shared drive, and then use tools such as NFS or Samba to provide remote connectivity and thus full "cloud builds". RATTLE stores three different types of information:

(1) **Command traces** record which files were accessed when a specific command line (including environment variables etc.) was run. For each file accessed, we record whether it was read or written to, and the hash of the file. These traces are used to determine whether to skip running a command as per §2.4, and are required to avoid continually rebuilding everything. These traces are each relatively small and there may be hundreds per run (one per command executed), meaning the cost to storing them is fairly low.

(2) **Speculation traces** record which commands were required by the last run. If the speculation information is missing, RATTLE uses an empty speculation trace, and won't get any speculative parallelism as per §2.8. There is only one speculation trace stored or queried per run, so the cost of these traces is very low.

(3) **File contents** record mappings from file hashes to the complete files. These hashes are used in conjunction with the command traces to allow copying outputs if all the inputs match, as per §2.5. Given a local file system with cheap copy-on-write semantics, it would be fairly cheap to add entries to the the file contents mapping. Unfortunately, copy-on-write is not a standard feature of most file systems, so copying files to this storage can have approximately a 10% performance cost.

When speculation traces are shared between multiple users, who may be compiling different versions of the software, it is useful to index the speculation trace by commit reference, so that the nearest ancestor commit with speculation information can be used. Because speculation information is necessarily approximate, using slightly old speculation information is not harmful. However, in practice we currently store only the latest speculation information, and because RATTLE is currently used with only small numbers of users, this approach works sufficiently well (but would definitely not scale to more users). Addressing this limitation is on our roadmap, and should not be difficult. Our preliminary experiments suggest that we can recover almost as much parallelism as MAKE on the initial build using this technique.

When sharing storage between multiple untrusted users, it is important to consider security implications. The file contents store is indexed by hash, and checking the hash on retrieval ensures the integrity of the data (assuming a strong hash function). The command traces are fundamentally relied upon, so should only be written to be trusted users. The speculation traces aren't relied upon by the build system, but a malicious speculation trace could cause a user to execute a malicious command, so should also be written only by trusted users. We have not yet implemented any security restrictions.

3.5 Tracing approaches

In §2.3 we assume the existence of *dependency tracing* which can, after a command completes, tell us which files that command read and wrote. Unfortunately, such an API is *not* part of the POSIX standard, and is not easily available on any standard platform. We reuse the file access tracing features that have been added to SHAKE [Mitchell 2020], which in turn are built on top of FSATRACE [Acereda 2019]. The techniques and limitations vary by OS:

- On **Linux** we use LD_LIBRARY_PRELOAD to inject a different C library into the process, which records file accesses, and then forwards functions on to the original C library. There are two main reasons we might fail to detect a file access. Firstly, the library we use (namely FSATRACE) doesn't support all C library functions, as there are a great number which perform file accesses (e.g. in addition to the traditional open, there is also a newer openat). Fortunately, while it is difficult to determine the full C library file API, FSATRACE has extensive coverage in practice, including at least 28 different file-related functions on Linux platforms.[3] Secondly,

---

[3]Many of these functions appear to be entirely undocumented, and required reverse-engineering by the FSATRACE developers.

the program might not dynamically load the C library – that is common in programs that invoke system calls directly (typically Go programs [Donovan and Kernighan 2015]) or have their C library statically linked to the binary. In future we plan to integrate BigBro [Roundy 2019a] as an alternative, which uses ptrace and doesn't rely on using a dynamic C library.

- On **Mac** we also use LD_LIBRARY_PRELOAD. While the technique works on a Mac, from Mac OS X 1.10 onwards system binaries can't be traced due to System Integrity Protection[4]. As an example of a program this impacts, the C/C++ compiler is typically installed as a system binary. It is possible to disable System Integrity Protection (but not recommended by Apple); or to use non-system binaries (e.g. those supplied by Nix [Dolstra et al. 2004]); or to copy the system binary to a temporary directory (which works provided the binary does not afterwards invoke another system binary). The library preload mechanism is implemented by Fsatrace and the copying system binaries trick on Mac is implemented in Shake.

- On **Windows** it is possible to hook the Kernel API, which can be used to detect when any files are accessed. Since this hook is at the kernel level, all programs are subject to it, and while it is important to cover all entry points, that list of entry points is well documented and quite minimal, so easy to cover entirely. Implementing the hook is difficult, particularly around 32bit v 64bit differences, as custom assembly language trampolines must be used. Furthermore, some antivirus products (incorrectly) detect programs using the kernel hook as viruses. Windows kernel hooking is available in Fsatrace, although without support for 32bit processes that spawn 64bit processes. In practice this limitation is rarely a problem (most binaries are now 64bit), often easily remedied (switch to the 64bit version), and possible to fix in Fsatrace if it was necessary (the engineering is difficult, but the technique is known and used by other tools).

- On all platforms, Fsatrace does not record access to directories (as opposed to files). Thus, if the kind of file-listing logic seen in Figure 1 was moved into a binary that needed to be traced, changes to the directory itself (such as adding new files) would not trigger a reexecution of the relevant command.

- On all platforms, any subcommands that are executed have their system accesses tracked and attributed to the parent process. As an example, gcc invokes as when compiling to transform textual assembly code to machine code. The tracing of gcc will include a read access to the as binary and a write to the .o file that as produces.

- Files which are both written and deleted before the command completes are considered as temporary files. They are reported by Fsatrace, but we ignore them in Rattle. Examples of such files are the temporary textual assembly code produced by gcc to give to as, which is also written in a temporary directory with a random filename.

In practice, none of the limitations have been overly problematic in the examples we have explored. We have encountered one bug in Fsatrace which led to builds erroring due to unsupported changes to glibc internals[5], and none which led to incorrect build results.

The consequence of a tracing tool not reporting a file access would be failing to re-run a build command when the true dependencies change, or speculating a command when it shouldn't be. To take a concrete example, the gRPC generator (named protoc) is written in Go. If a Rattle build features the command:

```
cmd "protoc hello.grpc --js_out=js"
```

That reads the file `hello.proto`, and writes the file `js/hello.js`, but on Linux and Mac Rattle does not know that. If the file `hello.proto` changes, Rattle will fail to rebuild `js/hello.js` causing it to produce an *incorrect result*. If the result is taken from a cloud cache (§2.5) it will not fetch `js/hello.js`. The dependency on the `protoc` binary itself *is* tracked as that is done by the operating system.

To resolve the issue, it is necessary to explicitly tell Rattle the dependencies, with:

```
cmd "protoc hello.grpc --js_out=js"
    (Read ["hello.grpc"]) (Write ["js/hello.js"])
```

Unfortunately, it is not immediately obvious *when* a tracing tool has returned incomplete dependencies, and manually specifying dependencies results in all the missing and incomplete dependency issues that are common in Makefile-based builds, which Rattle was designed to avoid.

We designed Rattle to work with the limitations of the best cross-platform tracing easily available – but that involves trade-offs. An enhanced, portable system would be a significant enabler for Rattle. Our wishlist for tracing would include a cross-platform API, precise detection times, detection as access happens, and executing code at interception points.

4 CORRECTNESS

Rattle's design relies on taking a sequence of commands from the user, and additionally running some commands speculatively. In this section we argue for the correctness of this approach with respect to the reference semantics of sequential evaluation. Throughout, we assume the commands themselves are pure functions from the inputs to their outputs (as reported by tracing) – without this property no build system more sophisticated than the shell script of §1 would be able to perfectly replicate the sequential model.

4.1 Hazards, formally

In §2.6 we introduced the notion of hazards through intuition and how they are used. Here we give more precise definitions:

Read-write hazard A build has a *read-write hazard* if a file is written to after it has been read. Given the limitations of tracing (see §3.5) we require the stronger property that for all files which are both read and written in a single build, the command which did the write must have finished before any commands reading it start executing, because we cannot assume the command finished writing to the file anytime before it completed.

Write-write hazard A build has a *write-write hazard* if a file is written to after it has already been written. Stated equivalently, a build has a write-write hazard if more than one command in a build writes to the same file.

Speculative-write-read hazard A build has a *speculative-write-read hazard* if a file is written to by a speculated command, then read from by a command in the build script, before the build script has required the speculated command.

4.2 Correctness properties

We now state and prove (semi-formally) several key correctness properties of our approach, particularly in the treatment of hazards. We define the following terms:

Command A command c reads from a set of files (c_r), and writes to a set of files (c_w). We assume the set of files are disjoint ($c_r \cap c_r \equiv \emptyset$), as per §2.4.

Deterministic command A command is deterministic if for a given set of files and values it reads, it always writes the same values to the same set of files. We assume commands are deterministic.

Fixed point A deterministic command will have no effect if none of the files it reads or writes have changed since the last time it was run. Since the command is deterministic, and its reads/writes are disjoint, the same reads will take place, the same computation will follow, and then the original values will be rewritten as they were before. We describe such a command as having *reached a fixed point*.

Correct builds If a build containing a set of commands C has reached a fixed point for every command c within the set, then it is a correct build system as per the definition in §2.6.

No hazards A build has no hazards if none of the hazards from §4.1 occur.

Input files We define the input files as those the build reads from but does not write to, namely:

$$\bigcup_{c \in C} c_r \setminus \bigcup_{c \in C} c_w$$

Output files We define the output files as those the build writes to, namely:

$$\bigcup_{c \in C} c_w$$

CLAIM 1 (COMPLETENESS). *If a set of commands C is run, and no hazards arise, then every command within C has reached a fixed point.*

Proof: A command c has reached a fixed point if none of the files in c_r or c_w have changed. Taking c_r and c_w separately:

(1) None of the files in c_w have changed. Within a set of commands C, if any file was written to by more than one command it would result in a write-write hazard. Therefore, if a command wrote to a file, it must have been the *only* command to write to that file.

(2) None of the files c_r have changed. For a read file to have changed would require it to be written to afterwards. If a file is read from then written to it is a read-write hazard, so the files c_r can't have changed.

As a consequence, all commands within C must have reached a fixed point.

CLAIM 2 (UNCHANGING BUILDS). *A build with deterministic control logic that completes with no hazards has reached a fixed point.*

Proof: The build's deterministic control logic means that beginning with the first command in the build, which is unchanged given the same set of input files, the command will write to the same set of output files and produce the same results. It follows that the subsequent command in the build will be unchanged as well since the results of the previous command did not change. It follows by induction that each of the following commands in the build will remain unchanged. Because the commands in the build will not have changed, it follows from the proof of Claim 1 that the commands will not change the values of any files, therefore, the build is at a fixed point.

CLAIM 3 (REORDERED BUILDS). *Given a script with no hazards when executed sequentially, with the same initial file contents, any other ordering of those commands that also has no hazards will result in the same terminal file contents.*

Proof: The proof of Claim 1 shows that any script with no hazards will result in a fixed point. We can prove the stronger claim, that for any filesystem state where all *inputs* have identical contents, there is only one fixed point for any ordering of the same commands that has no hazards. For each

command, it can only read from files that are in the inputs or files that are the outputs of other commands. Taking these two cases separately:

(1) For read files that are in the inputs, they aren't written to by the build system (or they wouldn't be in the inputs), and they must be the same in both cases (because the filesystem state must be the same for inputs). Therefore, changes to the inputs cannot have an effect on the command.

(2) For read files that are the outputs of other commands, they must have been written to *before* this command, or a read-write hazard would be raised.

Therefore, the first command that performs writes cannot access any writes from other commands, and so (assuming determinism) its writes must be a consequence of only inputs. Similarly, the second command can only have accessed inputs and writes produced by the first command, which themselves were consequences of inputs, so that commands writes are also consequences of the input. By induction, we can show that all writes are consequences of the inputs, so identical inputs results in identical writes.

CLAIM 4 (PARALLEL COMMANDS). *Given a script with no hazards when executed sequentially, any parallel or interleaved execution without hazards will also be equivalent.*

Proof: None of the proof of Claim 3 relies on commands not running in parallel, so that proof still applies.

CLAIM 5 (ADDITIONAL COMMANDS HAVE NO EFFECT). *Given a script with no hazards when executed, speculating unnecessary commands that do not lead to hazards will not have an effect on the build's inputs or outputs.*

Proof: Provided the inputs are unchanged, and there are no hazards, the proof from Claim 3 means the same outputs will be produced. If speculated commands wrote to the outputs, it would result in a write-write hazard. If speculated commands wrote to the inputs before they were used, it would result in a speculative-write-read hazard. If speculated commands wrote to the inputs after they were used, it would result in a read-write hazard. Therefore, assuming no hazards are raised, the speculated commands cannot have an effect on the inputs or outputs.

CLAIM 6 (PRESERVATION OF HAZARDS). *If a sequence of commands leads to a hazard, any additional speculative commands or reordering will still cause a hazard.*

Proof: All hazards are based on the observation of file accesses, not the absence of file accesses. Therefore, additional speculative commands do not reduce hazards, only increase them, as additional reads/writes can only cause additional hazards. For reordering, it cannot reduce write-write hazards, which are irrespective of order. For reordering of read-write hazards, it can only remove such a hazard by speculating the writing command before the reading command. However, such speculation is a speculative-read-write hazard, ensuring a hazard will always be raised.

4.3 Hazard Recovery

If a build involving speculation reaches a hazard there are several remedies that can be taken. In this section we outline these approaches and when each approach is safe to take.

Restart with no speculation. Restarting the entire build that happened with speculation, with no speculation, is safe and gives an equivalent result provided the inputs to the build have not changed (Claims 3, 5). Unfortunately, while writing to an input is guaranteed to give a hazard, restarting does *not* change the input back. As a concrete example, consider the build script with the

single command `gcc -c main.c`. If we speculate the command `echo 1 >> main.c` first, it will raise a speculative-write-read hazard, but restarting will not put `main.c` back to its original value, potentially breaking the build indefinitely. We discuss consequences and possible remediations from speculative commands writing to inputs in §3.3, but note such a situation should be very rare and likely easily reversible with minor user intervention. The build system only executes commands that were previously part of the build script, making it unlikely a build author would include commands that would irreversibly break their build. For all other recovery actions, we assume that restarting with no speculation is the baseline state, and consider them safe if it is equivalent to restarting with no speculation. This approach is the one we currently take in RATTLE; below we describe other approaches also justified in our model.

Restart with less speculation. Given the proofs in this section, running again with speculation is still safe. However, running with speculation might repeatedly cause the same speculation hazard, in which case the build would never terminate. Therefore, it is safe to restart with speculation provided there is strictly less speculation, so that (in the worst case) eventually there will be a run with no speculation.

As a practical matter, if a command is part of the commands that cause a hazard, it is a good candidate for excluding from speculation.

Raise a hazard. Importantly, if there is no speculation (including reordering and parallelism), and a hazard occurs, the hazard can be raised immediately. This property is essential as a base-case for restarting with no speculation.

If two required commands, that is commands in the users build script, cause a write-write hazard, and there have been no speculative-write-read hazards, then the same hazard will be reraised if the command restarts. For write-write, this inevitability is a consequence of Claim 3, as the same writes will occur. If two required commands cause a read-write hazard, and the relative order they were run in matches their required ordering, then the same hazard will be reraised if the command restarts, because the relative ordering will remain the same, Claim 6. However, if the commands weren't in an order matching their required ordering, it is possible the write will come first avoiding the hazard, so it cannot be immediately raised.

Continue. If the hazard only affects speculated commands, and those commands have no subsequent effect on required commands, it is possible to continue the build. However, it is important that the commands have no subsequent effect, even on required commands that have not yet been executed, or even revealed to the build system. Consequently, even if a build can continue after an initial hazard, it may still fail due to a future hazard. A build can continue if the hazards only affect speculated commands, that is either a *write-write hazard* where both commands were speculated, or a *read-write hazard* where the read was speculated. For other hazards, continuing is unsafe. If a speculated command is affected by an initial hazard, and that command is later required by the build script, a new non-continuable hazard will be raised.

5 EVALUATION

In this section we evaluate the design from §2, specifically our implementation from §3. We show how the implementation performs on the example from §1 in §5.1, on microbenchmarks in §5.2, and then on real projects that currently use MAKE – namely FSATrace (§5.3), Redis (§5.4), Vim (§5.5), tmux (§5.6), and Node.js (§5.7). We implement two sample projects using RATTLE in §5.9.

We have focused our real-world comparison on MAKE projects using C because such projects often have minimal external dependencies (making them easier to evaluate in isolation), and despite using the same programming language, have an interesting variety of dependency tracking strategies and

compilation times (ranging from one second to three hours). Most newer programming languages (e.g. Rust, Haskell etc) have dedicated built tools (e.g. Cargo, Cabal) so small projects in such languages tend not to use general-purpose build tools.

For benchmarks, the first four (§5.1-§5.4) were run on a 4 core Intel i7-4790 3.6GHz CPU (16Gb RAM, 7200 RPM rotating hard drive). The remaining benchmarks were run on a 32 core (2 threads per core) Intel(R) Xeon(R) CPU E5-4620 0 @ 2.20GHz (142Gb RAM, RAID 5 with 4 drives. RPM 10K SAS 6 Gbps).

5.1 Validating Rattle's suitability

In §1 we claimed that the following build script is "just as good" as a proper MAKE script.

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

There are two axes on which to measure "just as good" – correctness and performance. Performance can be further broken down into how much rebuilding is avoided, how much parallelism is achieved and how much overhead RATTLE imposes.

Correctness. RATTLE is correct, in that the reference semantics is running all the commands, and as we have shown in §2 (see §4 for detailed arguments), and tested for with examples, RATTLE obeys those semantics. In contrast, the MAKE version may have missing dependencies which causes it not to rebuild. Examples of failure to rebuild include if there are changes in gcc itself, or changes in any headers used through transitive #include which are not also listed in the MAKE script.

Rebuilding too much. RATTLE only reruns a command if some of the files it reads or writes have changed. It is possible that a command only depends on part of a file, but at the level of abstraction RATTLE works, it never rebuilds too much. As a matter of implementation, RATTLE detects changes by hashing the file contents, while MAKE uses the modification time. As a consequence, if a file is modified, but its contents do not change (e.g. using touch), MAKE will rebuild but RATTLE will not.

Parallelism. The script from §1 has three commands – the first two can run in parallel, while the the third must wait for both to finish. MAKE is given all this information by dependencies, and will always achieve as much parallelism as possible. In contrast, RATTLE has no such knowledge, so has to recover the parallelism by speculation (see §2.8). During the first execution, RATTLE has no knowledge about even which commands are coming next (as described in §2.2), so it has no choice but to execute each command serially, with less parallelism than MAKE. In subsequent executions RATTLE uses speculation to always speculate on the second command (as it never has a hazard with the first), but never speculate on the third until the first two have finished (as they are known to conflict). Interestingly, sometimes RATTLE executes the third command (because it got to that point in the script), and sometimes it speculates it (because the previous two have finished) – it is a race condition where both alternatives are equivalent. While RATTLE has less parallelism on the first execution, if we were to use shared storage for speculation traces, that can be reduced to the first execution *ever*, rather than the first execution for a given user.

Overhead. The overhead inherent in RATTLE is greater than that of MAKE, as it hashes files, traces command execution, computes potential hazards, figures out what to speculate and writes to a shared cloud store (in this section we use a locally mounted shared drive, as per §3.4). To measure the overhead, and prove the other claims in this section, we created a very simple pair of files, main.c and util.c, where main.c calls printf using a constant returned by a function in util.c. We then measured the time to do: (1) An initial build from a clean checkout; (2) a rebuild

when nothing had changed; (3) a rebuild with whitespace changes to main.c, resulting in the same main.o file; (4) a rebuild with meaningful changes to main.c, resulting in a different main.o file; and (5) a rebuild with meaningful changes to both C files.

We performed the above steps with 1, 2 and 3 threads, on Linux. To make the parallelism obvious, we modified gcc to sleep for 1 second before starting. The build times are (lower is better):

| Number of threads | 1 | | 2 | | 3 | |
Tool	MAKE	RATTLE	MAKE	RATTLE	MAKE	RATTLE
1) Initial build	3.35s	3.28s	2.18s	3.28s	2.20s	3.36s
2) Nothing changed	0.00s	0.00s	0.00s	0.00s	0.00s	0.00s
3) main.c changed whitespace	2.19s	1.12s	2.19s	1.12s	2.19s	1.10s
4) main.c changed	2.19s	2.16s	2.23s	2.21s	2.19s	2.22s
5) Both C files changed	3.28s	3.29s	2.22s	2.20s	2.19s	2.20s

As expected, we see that during the initial build RATTLE doesn't exhibit any parallelism, but MAKE does (1). In contrast, RATTLE benefits when a file changes in whitespace only and the resulting object file doesn't change, while MAKE can't (3). We see 3 threads has no benefit over 2 threads, as this build contains no more parallelism opportunities. Comparing the non-sleep portion of the build, MAKE and RATTLE are quite evenly matched, typically within a few milliseconds, showing low overheads. We focus on the overheads in the next section.

5.2 Measuring overhead

In order to determine what overhead RATTLE introduces, we ran a fixed set of commands with increasingly more parts of RATTLE enabled. RATTLE command execution builds on the command execution from SHAKE [Mitchell 2012], which uses FSATRACE for tracing and the Haskell process library for command execution. We ran the commands in a clean build directory in 7 ways:

(1) Using make -j1 (MAKE with 1 thread), as a baseline.
(2) Using System.Process from the Haskell process library.
(3) Using cmd from the SHAKE library [Mitchell 2012], which builds on the process library.
(4) Using cmd from SHAKE, but wrapping the command with FSATRACE for file tracing.
(5) Using cmd from SHAKE with the Traced setting [Mitchell 2020], which runs FSATRACE and collects the results.
(6) Using RATTLE with no speculation or parallelism, and without shared storage.
(7) Using RATTLE with all features, including shared storage.

To obtain a set of commands typical of building, we took the latest version of FSATRACE[6] and ran make -j1 with maximum verbosity, recording the commands that were printed to the standard output. On Windows FSATRACE runs 25 commands (21 compiles, 4 links). On Linux FSATRACE runs 9 commands (7 compiles, 2 links). On Linux the list of commands produces write-write hazards, because it compiles some files (e.g. shm.c) twice, once with -fPIC (position independent code), and once without. However, both times it passes -MMD to cause gcc to produce shm.d at the same location (shm.d is used to track dependencies). That write-write hazard is a genuine problem, and in an incredibly unlucky build FSATRACE would end up with a corrupted shm.d file. To remedy the problem we removed the -MMD flag as it doesn't impact the benchmark and isn't required by RATTLE. We ran all sets of commands on both Windows and Linux.

---

[6]https://github.com/jacereda/fsatrace/commit/41fbba17da580f81ababb32ec7e6e5fd49f11473

Commands	Windows			Linux		
1) Make	9.96s	100%		1.19s	100%	
2) process	10.26s	103%	+3%	1.18s	99%	-1%
3) Shake	10.58s	106%	+3%	1.17s	98%	-1%
4) Shake + Fsatrace	12.66s	127%	+21%	1.23s	103%	+5%
5) Shake + Traced	13.06s	131%	+4%	1.23s	103%	+0%
6) Rattle	14.43s	145%	+14%	1.25s	105%	+2%
7) Rattle + everything	14.53s	146%	+1%	1.27s	107%	+2%

In the above table both Windows and Linux have three columns – the time taken (average of the three fastest of five runs), that time as a percentage of the baseline Make run, and the delta from the row above. The results are significantly different between platforms:

Windows. On Windows, we see that the total overhead of Rattle makes execution 46% slower. 21% of the slowdown is from Fsatrace (due to hooking Windows kernel API), with the next greatest overhead being from Rattle itself. Of the Rattle overhead, the greatest slowdown is caused by canonicalizing paths. Using the default NTFS file system, Windows considers paths to be case insensitive. As a result, we observe paths like `C:\windows\system32\KERNELBASE.dll`, which on disk are called `C:\Windows\System32\KernelBase.dll`, but can also be accessed by names such as `C:\Windows\System32\KERNEL~1.DLL`. Unfortunately, Windows also supports case sensitive file systems, so using case-insensitive equality is insufficient.

On Windows, enabling the default anti-virus (Windows Defender) has a significant impact on the result, increasing the Make baseline by 11% and the final time by 29%. These results were collected with the anti-virus disabled.

Linux. On Linux, the total overhead is only 7%, of which nearly all (5%) comes from tracing. The initial Haskell process creation is faster than direct Make execution (by 1%) because the overheads of the Haskell process library on Linux are less than those of Make doing dependency checking and process execution.

These results show that tracing has a minor but not insignificant effect on Linux, whereas on Windows can be a substantial performance reduction. Additionally, the Windows results had far more variance, including large outliers. Thus, our remaining benchmarks were all run on Linux.

5.3 Fsatrace

To compare Make and Rattle building Fsatrace we took the commands we extracted for §5.2 and ran the build script for the 100 previous commits in turn, starting with a clean build then performing incremental builds. To make the results more readable, we hid any commits where all builds took < 0.02s, resulting in 12 interesting commits. We ran with 1 to 4 threads, but omit the 2 and 3 thread case as they typically fall either on or just above the 4 thread results.

Compile time at each successive commit
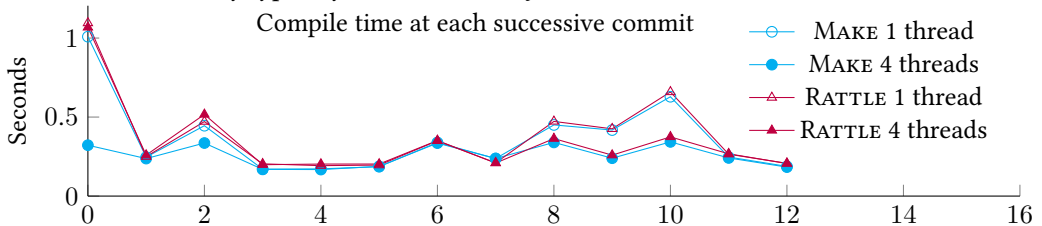
As we can see, the first build is always > 1s for Rattle, but Make is able to optimise it as low as 0.33s with 4 threads. Otherwise, Rattle and Make are competitive – users would struggle to see

the difference. The one commit that does show some variation is commit 2, where both are similar for 1 thread, but RATTLE at 4 threads is slightly slower than 1 thread, while MAKE is faster. The cause is speculation leading to a write-write hazard. Concretely, the command for linking fsatrace.so changed to include a new file proc.o. RATTLE starts speculating on the old link command, then gets the command for the new link. Both commands write to fsatrace.so, leading to a hazard, and causing RATTLE to restart without speculation.

5.4 Redis
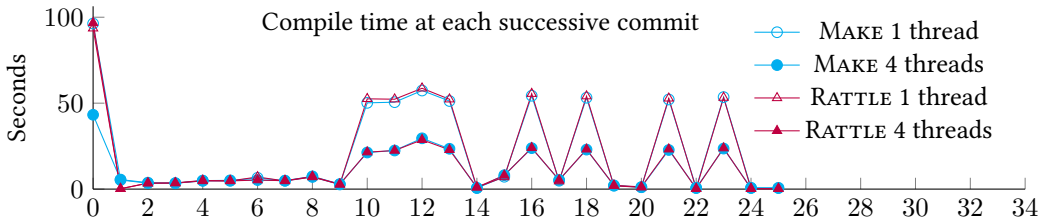
Redis is an in-memory NoSQL database written in C, developed over the last 10 years. Redis is built using recursive MAKE [Miller 1998], where the external dependencies of Redis (e.g. Lua and Linenoise) have their own MAKE scripts which are called by the main Redis MAKE script. Using stamp files, on successive builds, the external dependencies are not checked – meaning that users must clean their build when external dependencies change.

When benchmarking RATTLE vs MAKE we found that RATTLE was about 10% slower due to copying files to the file contents cloud cache (see §3.4), something MAKE does not do. Therefore, for a fairer comparison, we disabled copying these files, but continue to store command and speculation traces. For local builds, the consequence of disabling the cloud cache is that if a file changes, then changes back to a previous state, we will have to rebuild rather than get a cache hit – something that never happens in this benchmark. With those changes RATTLE is able to perform very similarly to MAKE, despite properly checking external dependencies. Differences occur on the initial build, where RATTLE has no information to speculate with, and commit 1, where RATTLE correctly detects no work is required but MAKE rebuilds some files.

5.5 Vim
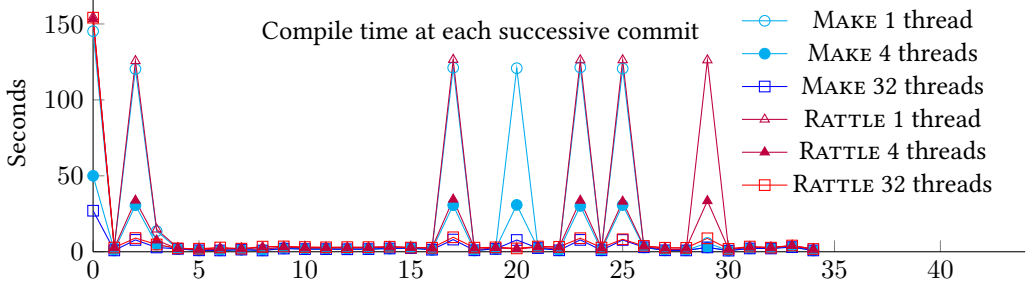
Vim is a popular text editor, over 25 years old. The majority of the source code is Vim script and C, and it is built with MAKE. To compare the original MAKE based build to RATTLE, we built Vim over 35 commits, using both the original MAKE and a RATTLE script created by recording the commands MAKE executed on a clean build. The benchmarks show that MAKE and RATTLE had similar performance for almost every commit, apart from two.

(1) For the 20th commit in the chart, MAKE did much more work than RATTLE. This commit changed the script which generates osdef.h, in such a way that it did not change the contents

of osdef.h. MAKE rebuilt all of the object files which depend on this header file, while RATTLE skipped them as the file contents had not changed.

(2) For the 29th commit in the chart, RATTLE did much more work than MAKE. This commit changed a prototype file that is indirectly included by all object files. This prototype file is not listed as a dependency of the object files, so MAKE did not rebuild them, but RATTLE noticed the changed dependency. This missing dependency could have resulted in an incorrect build or a build error, but in this case (by luck) seems not to have been significant. A future prototype change may result in incorrect builds.

5.6 tmux
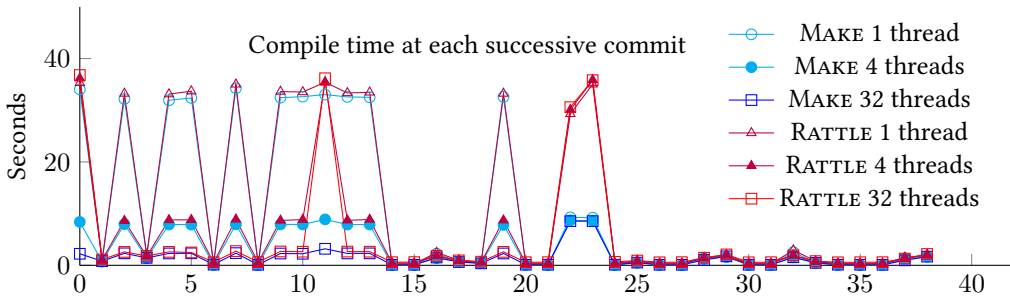
tmux is a terminal multiplexer in development for the last 12 years. It is built from source using `sh autogen.sh && ./configure && make` sequence, making use of AUTOCONF, AUTOMAKE, PKG-CONFIG, and MAKE. To compare a RATTLE based tmux build to the MAKE based one, we first ran `sh autogen.sh && ./configure`, and then used the resulting MAKE scripts to generate RATTLE build scripts. tmux was built over a series of 39 commits with both MAKE and RATTLE. For the 11th, 22nd and 23rd commits in the chart, RATTLE and MAKE took noticeably different amounts of time. In all cases the cause was a build script change, preventing RATTLE from speculating, and thus taking approximately as long as a full single-threaded build. Additionally, a write-write hazard occurred at each of those commits caused by a conflict between a speculated command and its new version in the changed script.
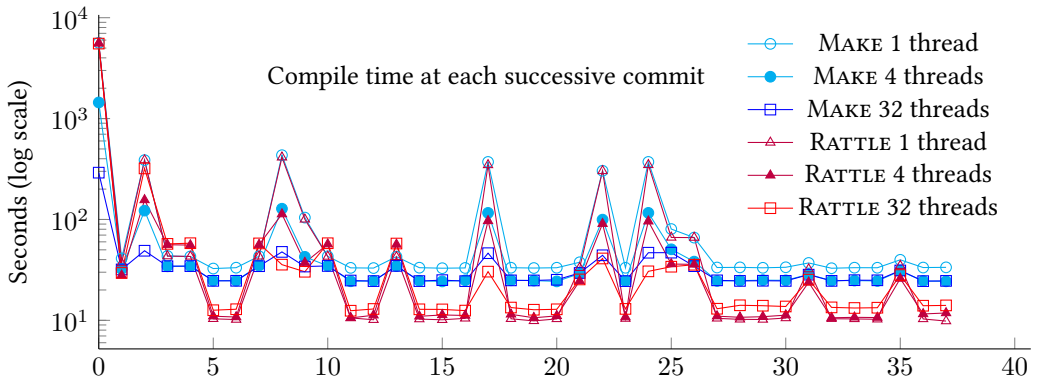
5.7 Node.js

Node.js is a JavaScript runtime built on Chrome's V8 Javascript engine that has been in development since 2009. The project is largely written in JavaScript, C++, Python and C, and is built using MAKE and a meta-build tool called Generate Your Projects (GYP) [Google 2009].

Node.js build system. To build Node.js from source, a user first runs `./configure`, which configures the build and runs GYP, generating Makefiles. From then on, a user can build with MAKE. GYP generates the majority of the Makefiles used to build Node.js, starting from a series of source `.gyp` files specifying metadata such as targets and dependencies. GYP generates a separate `*.mk` Makefile for each target, all of which are included by a generated top-level Makefile. That top-level Makefile can be run by MAKE, but through a number of tricks, GYP controls the dependency calculation:

(1) All tracked commands are run using the `do_cmd` function from the top-level Makefile. The `do_cmd` function checks the command, and if anything has changed since the last run, reruns it. To ensure this function is called on every run, targets whose rules use `do_cmd` have a fake dependency on the phony target `FORCE_DO_CMD`, forcing MAKE to rerun them every time. This mechanism operates much like RATTLE command skipping from §2.4.

(2) Tracked commands create dependency files for each target, recording observed dependencies from the last run. Usually this information is produced by the compiler, e.g. `gcc -MMD`.

(3) Generated source files are produced and managed using `.intermediate` targets. By relying on a combination of `do_cmd`, `FORCE_DO_CMD` and a special prerequisite `.INTERMEDIATE` that depends on all `.intermediate` targets, the intention appears to be to regenerate the generated code whenever something relevant has changed. However, in our experiments, those intermediate targets seem to run every time.

It appears the build of Node.js has tried to take an approach closer to that of RATTLE, namely, tracking dependencies each time a command runs and making sure commands re-run when they have changed. The implementation is quite complicated and relies on deep knowledge of precisely how MAKE operates, but appears to not quite match what was intended.

To compare to RATTLE, we built Node.js over a series of 38 commits with both MAKE and RATTLE, with 1, 4, and 32 threads. The RATTLE build was created for each commit by recording every command executed by the original build, excluding the commands exclusively dealing with dependency files. Due to the large variations in time, the above plot uses a log scale.

For building from scratch, MAKE and RATTLE took approximately the same amount of time with a single thread (around 94 minutes). However, with 4 and 32 threads, MAKE was able to build Node nearly four times faster (24 minutes), and approximately 20 times faster (4.8 minutes), respectively, while RATTLE had no speculation information and did not improve. When built from scratch with 4 and 32 threads, the RATTLE build was not meaningfully faster (92-93 minutes). For many commits RATTLE and MAKE give similar results, but for those with the least amount of work, where only a few files change, RATTLE is able to beat MAKE. The fastest observed MAKE time across all thread counts is approximately 25 seconds, while RATTLE goes as low as 10 seconds. The major cause for the slowdown in MAKE is that MAKE *always* rebuilds the generated files but RATTLE (correctly) does not.

While building the third commit (labeled 2 in chart), a write-write hazard occurred, which caused the build to re-run with no speculation. But, as we can see from the graph, when built with 4 threads even the failed speculation was still faster than the sequential version. However, the time it took RATTLE to build the third commit with 32 threads was almost as long as it took to build it sequentially, probably due to thread overhead or unfortunate scheduling leading to the write-write hazard sooner in the build. While building commits 4, 5, 8, 11, 14, read-write hazards occurred which caused the build to re-run these commits with no speculation, however, these hazards do not appear to have greatly impacted the build performance.

5.8 Summary comparing with MAKE

Looking across the projects in this section, a few themes emerge.

- For most commits, RATTLE and MAKE give similar performance. The one case that is not true is the initial build at higher levels of parallelism, where MAKE outperforms RATTLE, because RATTLE has no speculation information. If RATTLE used a shared speculation cache as per §3.4, where another user had already built the initial commit, the first build would be no different as it would also have good speculation information.
- The difference between 1 thread and 4 threads can be quite significant, and typically MAKE and RATTLE do equally well with parallelism. The implicit parallelism in RATTLE is an important feature which works as designed.
- Hazards are relatively rare, but when they do occur, they do impact the build time.
- Every MAKE project investigated manages dependencies differently. Most use gcc to generate dependency information. Some provide that information directly (e.g. FSATRACE), some use stamp files (e.g. Redis) and some build their own mechanisms on top of MAKE (e.g. Node.js, but also Mokhov et al. [2016, §2]).
- Many projects give up on dependencies in less common cases, requiring the user to manually invoke a clean, e.g. Redis for external dependencies, Vim for prototype changes.
- RATTLE correctly rebuilds if the build file itself updates, but most of the MAKE projects do not. The one exception is Node.js, through its use of its dependency encoding in the Makefile.

5.9 Writing RATTLE scripts

In order to see how difficult it is to write build scripts for RATTLE, we tried writing two examples – namely a build script FSATRACE, and a program to fetch Haskell dependencies and install them.

5.9.1 Building FSATRACE. We implemented a RATTLE script to build FSATRACE. Unlike §5.3, where we took the outputs from MAKE, this time we took the source Makefiles and tried to port their logic to RATTLE. We did not port the testing or clean phony target, but we did port both Windows and Linux support. The script first defines 11 variables, following the same pattern as the Makefile which defines the same variables – some of which are conditional on the OS. Next we compile all the C sources, and finally we do a few links. The total code is 19 non-comment lines, corresponding to 46 non-comment lines in the original Makefiles.

Given the setup of the code, it was trivial to use explicit parallelism for compiling all C files, but doing so for the linking steps, and arranging for the links to be performed as soon as possible, would have been tedious. Taking some representative pieces from the build, we have:

```
let cppflags | isWindows = "-D_WIN32_WINNT=0x600 -isysteminclude/ddk"
             | otherwise = "-D_GNU_SOURCE -D_DEFAULT_SOURCE=1"
let sosrcs = ["src/unix/fsatraceso.c","src/unix/shm.c","src/unix/proc.c"]

forP_ (srcs ++ if isWindows then dllsrcs else sosrcs) $ \x ->
    cmd "gcc -c" ["-fPIC" | not isWindows] cppflags cflags x "-o" (x -<.> "o")

cmd "gcc" ldflags ldobjs (map (-<.> "o") srcs) ldlibs "-o" ("fsatrace" <.> exe)
```

The overall experience of porting the Makefile was pleasant. The explicit isWindows is much nicer than the Makefile approach. The availability of lists and a rich set of operations on them made certain cases more natural. The use of functions, e.g. objects, allowed simplifying transformations and giving more semantically meaningful names. The powerful syntax of cmd, allowing multiple arguments, makes the code look relatively simple. The only downside compared to MAKE was the use of quotes in more places.

5.9.2 Installing Haskell Dependencies. As a second test, we wrote a program to fetch and build a Haskell library and its dependencies. For Haskell packages, Cabal is both a low-level library/interface

for configuring, building and installing, and also a high-level tool for building sets of packages. Stackage is a set of packages which are known to work together. Stack is a tool that consumes the list of Stackage packages, and using low-level Cabal commands, tries to install them. We decided to try and replicate Stack, with the original motivation that by building on top of RATTLE we could get cloud builds of Haskell libraries for free.

We were able to write a prototype of something Stack-like using RATTLE in 71 non-comment lines. Our prototype is capable of building some projects, showing the fundamental mechanisms work, but due to the number of corner cases in Haskell packages, is unlikely to be practical without further development. The main logic that drives the process is reproduced below.

```
stack :: String -> [PackageName] -> Run ()
stack resolver packages = do
    let url = "https://www.stackage.org/" ++ resolver ++ "/cabal.config"
    cmd Shell "curl -sSL" url "-o" resolver <.> "config"
    versions <- liftIO $ readResolver $ resolver <.> "config"
    flags <- liftIO extraConfigureFlags
    needPkg <- memoRec $ \needPkg name -> do
        let v = version Map.! name
        whenJust v $ installPackage needPkg config flags name
        return v
    forP_ packages needPkg
```

Reading this code, given a Stackage resolver (e.g. nightly-2012-10-29), we download the resolver to disk, read it, then recursively build projects in an order satisfying dependencies. The download uses curl, which has no meaningful reads, so will always be skipped. Importantly, the URL (and thus the command line) includes the full resolver which will never change once published. The parsing of the resolver is Haskell code, not visible to RATTLE, repeated on every run (the time taken is negligible). We then build a recursive memoisation pattern using memoRec, where each package is only built once, and while building a package you can recursively ask for other packages. We then finally demand the packages that were requested.

Writing the program itself was complex, due to Haskell packaging corner cases – but the RATTLE aspect was pleasant. Having Haskell available to parse the configuration and having efficient Map data structures made it feasible. Not worrying about dependencies, focusing on it working, vastly simplified development. There were lots of complications due to the nature of Haskell packages (e.g. some tools/packages are special, parsing metadata is harder than it should be), but only a few which interacted closely with RATTLE:

(1) There are some lock files, e.g. package.cache.lock, which are both read and written by many commands, but not in an interesting way. We explicitly ignored these files from being considered for hazards.

(2) Haskell packages are installed in a database, which by default is appended to with each package registration. That pattern relies on the order of packages and is problematic for concurrency. Many build systems, e.g. BAZEL [Google 2016], use multiple package databases for Haskell, each containing one package. We follow the same pattern.

(3) By default, Cabal produces packages where some fields (but not all) include an absolute path. To make the results generally applicable for cloud builds, we remove the absolute paths. To ensure the absolute paths never make it to the cloud, we run the command that produces and the command that cleans up as a single command from the point of view of RATTLE.

(4) Since cachable things have to be commands, that required putting logic into commands, not just shell calls. To make that pleasant, we wrote a small utility to take Template Haskell

commands [Sheard and Peyton Jones 2002] and turn them into Haskell scripts that could then be run from the command line.

6 RELATED WORK

The vast majority of existing build systems are *backward build systems* – they start at the final target, and recursively determine the dependencies required for that target. In contrast, RATTLE is a *forward build system*—the script executes sequentially in the order given by the user.

6.1 Comparison to forward build systems

The idea of treating a script as a build system, omitting commands that have not changed, was pioneered by MEMOIZE [McCloskey 2008] and popularised by FABRICATE [Hoyt et al. 2009]. In both cases the build script was written in Python, where cheap logic was specified in Python and commands were run in a traced environment. If a traced command hadn't changed inputs since it was last run, then it was skipped. We focus on FABRICATE, which came later and offered more features. FABRICATE uses strace on Linux, and on Windows uses either file access times (which are either disabled or very low resolution on modern Windows installations) or a proprietary and unavailable tracker program. Parallelism can be annotated explicitly, but often is omitted.

These systems did not seem to have much adoption – in both cases the original sources are no longer available, and knowledge of them survives only as GitHub copies. RATTLE differs from an engineering perspective by the tracing mechanisms available (see §3.5) and the availability of cloud build (see §2.5) – both of which are likely just a consequence of being developed a decade later. RATTLE extends these systems in a more fundamental way with the notion of hazards, which both allows the detection of bad build scripts, and allows for speculation – overcoming the main disadvantage of earlier forward build systems. Stated alternatively, RATTLE takes the delightfully simple approach of these build systems, and tries a more sophisticated execution strategy.

Recently there have been three other implementations of forward build systems we are aware of.

(1) SHAKE [Mitchell 2012] provides a forward mode implemented in terms of a backwards build system. The approach is similar to the FABRICATE design, offering skipping of repeated commands and explicit parallelism. In addition, SHAKE allows caching custom functions as though they were commands, relying on the explicit dependency tracking functions such as need already built into SHAKE. The forward mode has been adopted by a few projects, notably a library for generating static websites/blogs. The tracing features are provided by a combination of SHAKE and FSATRACE, and are the ones we reuse in RATTLE.

(2) FAC [Roundy 2019b] is based on the BIGBRO tracing library. Commands are given in a custom file format as a static list of commands (i.e. no monadic expressive power as per §2.2), but may optionally include a subset of their inputs or outputs. The commands are not given in any order, but the specified inputs/outputs are used to form a dependency order which FAC uses. If the specified inputs/outputs are insufficient to give a working order, then FAC will fail but record the *actual* dependencies which will be used next time – a build with no dependencies can usually be made to work by running FAC multiple times.

(3) STROLL [Mokhov 2019] takes a static set of commands, without either a valid sequence or any input/output information, and keeps running commands until they have all succeeded. As a consequence, STROLL may run the same command multiple times, using tracing to figure out what might have changed to turn a previous failure into a future success. STROLL also reuses the tracing features of SHAKE and FSATRACE.

There are significantly fewer forward build systems than backwards build systems, but the interesting dimension starting to emerge is how an ordering is specified. The three current alternatives

are the user specifies a valid order (FABRICATE and RATTLE), the user specifies partial dependencies which are used to calculate an order (FAC) or the user specifies no ordering and search is used (STROLL and some aspects of FAC).

Other build systems, such as PLUTO [Erdweg et al. 2015] (and PIE [Konat et al. 2018a,b], a front-end to PLUTO), simulate some of the convenience of a forward build system by constructing the dependency graph during the execution of a host-language program that contains dependency information. However, these systems, similar to SHAKE, then rely on that dependency graph for rebuilding. When PLUTO performs an incremental build it must traverse the entire dependency graph to discover which tasks must be re-run. Building on PLUTO, PIE [Konat et al. 2018a,b] sought to increase the scalability of PLUTO's rebuilding algorithm by only traversing tasks transitively affected by changed files rather than traversing the entire dependency graph. In contrast, RATTLE and other forward build systems do not construct or use such a dependency graph. When RATTLE runs a build there is no dependency graph to traverse, and each command in the build is considered. Therefore, a rebuild scales with the size of the build, and the approach used in [Konat et al. 2018a] cannot be applied to RATTLE. Additionally, we have not found computing internal datastructures to be a bottleneck and have not attempted to avoid it.

6.2 Comparison to backward build systems

The design space of backward build systems is discussed in [Mokhov et al. 2018]. In that paper it is notable that forward build systems do not naturally fit into the design space, lacking the features that a build system requires. We feel that omission points to an interesting gap in the landscape of build systems. We think that it is likely forward build systems could be characterised similarly, but that we have yet to develop the necessary variety of forward build systems to do so. There are two dimensions used to classify backward build systems:

Ordering Looking at the features of RATTLE, the ordering is a sequential list, representing an excessively strict ordering given by the user. The use of speculation is an attempt to weaken that ordering into one that is less linear and more precise.

Rebuild For rebuilding, RATTLE looks a lot like the constructive trace model – traces are made, stored in a cloud, and available for future use. The one wrinkle is that a trace may be later invalidated if it turns out a hazard occurred (see §3.3).

There are three features present in some backward build systems that are particularly relevant to RATTLE:

Sandboxing Some backward build systems (e.g. BAZEL [Google 2016]) run processes in a sandbox, where access to files which weren't declared as dependencies are blocked – ensuring dependencies are always sufficient. A consequence is that it can be harder to write a BAZEL build script, requiring users to declare dependencies like gcc and system headers that are often overlooked. The sandbox doesn't prevent the reverse problem of too many dependencies.

The NIX build system [Dolstra et al. 2004] solves the problem of sandboxing in a slightly different way. NIX is a build tool aimed at package management, replacing tools such as apt and RPM. Before running any binary NIX rewrites the $PATH to remove everything apart from declared dependencies, ensuring no undeclared dependencies can be accessed. NIX also installs each package into a location which includes a cryptographic hash of its contents, ensuring that even packages with the same name end up at different locations unless they are identical.

Remote Execution Some build systems (e.g. BAZEL and BUILDXL [Microsoft 2020]) allow running commands on a remote machine, usually with a much higher degree of parallelism than is available on the users machine. If RATTLE was able to leverage remote execution then speculative commands could be used to fill up the cloud cache, and *not* cause local writes to disk, eliminating

all speculative hazards – a very attractive property. Remote execution in Bazel sends all required files along with the command, but since Rattle doesn't know the files accessed in advance, that model is infeasible. Remote execution in BuildXL sends the files it thinks the command will need, augments the file system to block if other files are accessed, and sends requests for additional files back to the originating machine – which would fit nicely with Rattle.

Hazards Some build systems detect when certain types of hazard occur. For example, Pluto [Erdweg et al. 2015] builds are constructed from builders and it is a requirement that no two builders generate the same file, and if they do the build is aborted. Similarly, if two commands in a Rattle build write to the same file a write-write hazard occurs and the build is terminated. If a command in a Rattle build writes to a file after another command has already read it a read-write hazard occurs. Analogously in Pluto, if a builder requires a file generated by another builder, then the builder which generated the file must be required first and if it is not the build is aborted.

6.3   Build correctness

We aren't the first to observe that existing build systems often have incorrect dependencies. Bezemer et al. [2017] performed an analysis of the missing dependencies in Make build scripts, finding over *1.2 million unspecified dependencies* among four projects. To detect missing dependencies, Licker and Rice [2019] introduced a concept called *build fuzzing*, finding race-conditions and errors in 30 projects. It has also been shown that build maintenance requires as much as a 27% overhead on software development [McIntosh et al. 2011], a substantial proportion of which is devoted to dependency management. Our anecdotes from §5 all reinforce these messages.

6.4   Speculation

Speculation is used extensively in many other areas of computer science, from processors to distributed systems. If an action is taken before it is known to be required, that can increase performance, but undesired side-effects can occur. Most speculating systems attempt to block the side-effects from happening, or roll them back if they do.

The most common use of speculation in computer science is the CPU – Swanson et al. [2003] found that 93% of useful CPU instructions were evaluated speculatively. CPUs use hazards to detect incorrect speculation, with similar types of read/write, write/write and write/read hazards [Patterson and Hennessy 2013] – our terminology is inspired by their approaches. For CPUs many solutions to hazards are available, e.g. stalling the pipeline if a hazard is detected in advance or the Tomasulo algorithm [Tomasulo 1967]. Rattle also stalls the pipeline (stops speculating) if it detects potential hazards, although does so with incomplete information, unlike a CPU. The Tomasulo algorithm involves writing results into temporary locations and the moving them over afterwards – use of remote execution (§6.2) might be a way to apply similar ideas to build systems.

Looking towards software systems, Welc et al. [2005] showed how to add speculation to Java programs, by marking certain parts of the program as worth speculating with a future. Similar to our work, they wanted Java with speculation to respect the semantics of the sequential version of the program, which required two main techniques. First, all data accesses to shared state are tracked and recorded, and if a dependency violation occurs, the offending code is restarted. Second, shared state is versioned using a copy-on-write invariant to ensure threads write to their own copies, preventing a future from seeing its continuation's writes.

Thread level speculation [Steffan and Mowry 1998] is used by compilers to automatically paral-lelise programs, often by executing multiple iterations of a loop body simultaneously. As before, the goal is to maintain the semantics of single-threaded execution. Techniques commonly involve buffering speculative writes [Steffan et al. 2000] and ensuring that a read reflects the speculative writes of threads that logically precede it.

Speculation has also been investigated for distributed systems. Nightingale et al. [2005] showed that adding speculation to distributed file systems such as NFS can make some benchmarks over 10 times faster, by allowing multiple file system operations to occur concurrently. A model allowing more distributed speculation, even in the presence of message passing between speculated distributed processes, is presented by Tapus [2006]. Both these pieces of work involve modifying the Linux kernel with a custom file system to implement roll backs transparently.

All these approaches rely on the ability to trap writes, either placing them in a buffer and applying them later or rolling them back. Unfortunately, such facilities, while desirable, are currently difficult to achieve in portable cross-platform abstractions (§3.5). We have used the ideas underlying speculative execution, but if the necessary trapping/rollback facilities became available in future, it's possible we could follow the approaches more directly.

7 CONCLUSION AND FUTURE WORK

In this paper we present Rattle, a build system that takes a sequence of actions and treats them as a build script. From the user perspective, they get the benefits of a conventional build system (incrementality, parallelism) but at lower cost (less time thinking about dependencies, only needing to supply a valid ordering). Compared to conventional build systems like Make, Rattle presents a simpler user interface (no dependencies), but a more complex implementation model.

Our evaluation in §5 shows that for some popular real-world projects, switching to Rattle would bring about simplicity and correctness benefits, with negligible performance cost. The two places where builds aren't roughly equivalent to Make are the initial build (which can be solved with a global shared cache) and when speculation leads to a hazard. There are several approaches to improving speculation, including giving Rattle a list of commands that should not be speculated (which can be a perfect list for non-monadic builds), or giving Rattle a subset of commands' inputs/outputs (like Fac does), or implementing better recovery strategies from speculation errors.

Our evaluation focuses on projects whose build times are measured in seconds or minutes, with only one project in the hours range. It is as yet unclear whether similar benefits could be achieved on larger code bases, and whether the Rattle approach of "any valid ordering" is easy to describe compositionally for large projects. But, it does seem clear that existing projects built with Make seek some form of automatic dependency detection, with most projects using some form of gcc dependency generation manually wired into the build system.

Our next steps are scaling Rattle and incorporating feedback from actual users.

REFERENCES

Jorge Acereda. 2019. FSATrace: filesystem access tracer. (2019). https://github.com/jacereda/fsatrace.

Lennart Augustsson, Mikael Rittri, and Dan Synek. 1994. On generating unique names. *JFP* 4 (01 1994), 117–123.

Cor-Paul Bezemer, Shane McIntosh, Bram Adams, Daniel German, and Ahmed Hassan. 2017. An empirical study of unspecified dependencies in Make-based build systems. *Empirical Software Engineering* 22, 6 (2017), 3117–3148.

Eelco Dolstra, Merijn De Jonge, Eelco Visser, and others. 2004. Nix: A Safe and Policy-Free System for Software Deployment. In *LISA*, Vol. 4. 79–92.

Alan Donovan and Brian Kernighan. 2015. *The Go Programming Language* (1st ed.). Addison-Wesley Professional.

Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015. A Sound and Optimal Incremental Build System with Dynamic Dependencies. In *Proceedings of OOPSLA 2015*. ACM, 89–106. DOI:http://dx.doi.org/10.1145/2814270.2814316

Stuart Feldman. 1979. Make - A program for maintaining computer programs. *Software: Practice and experience* 9, 4 (1979), 255–265.

Google. 2009. Generate Your Projects. (2009). https://gyp.gsrc.io/.

Google. 2016. Bazel. (2016). http://bazel.io/.

Berwyn Hoyt, Bryan Hoyt, and Ben Hoyt. 2009. Fabricate: The better build tool. (2009). https://github.com/SimonAlfie/fabricate.

Mark Jones. 1995. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. 97–136.

Oleg Kiselyov. 2015. Polyvariadic functions and keyword arguments: pattern-matching on the type of the context. (2015). http://okmij.org/ftp/Haskell/polyvariadic.html.

Gabriël Konat, Sebastian Erdweg, and Eelco Visser. 2018a. Scalable Incremental Building with Dynamic Task Dependencies. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 76âĂŞ86. DOI:http://dx.doi.org/10.1145/3238147.3238196

Gabriël Konat, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. 2018b. PIE: A Domain-Specific Language for Interactive Software Development Pipelines. *The Art, Science, and Engineering of Programming* 2, 3 (Mar 2018). DOI:http://dx.doi.org/10.22152/programming-journal.org/2018/2/9

Nándor Licker and Andrew Rice. 2019. Detecting incorrect build rules. In *Proceedings of ICSE 2019*. ACM, 1234–1244. DOI:http://dx.doi.org/10.1109/ICSE.2019.00125

Bill McCloskey. 2008. Memoize. (2008). https://github.com/kgaughan/memoize.py.

Shane McIntosh, Bram Adams, Thanh Nguyen, Yasutaka Kamei, and Ahmed Hassan. 2011. An empirical study of build maintenance effort. In *Proceedings of ICSE 2011*. ACM, 141–150.

Microsoft. 2020. Build Accelerator. (2020). https://github.com/microsoft/BuildXL.

Peter Miller. 1998. Recursive Make considered harmful. *Journal of AUUG Inc* 19, 1 (1998), 14–25.

Neil Mitchell. 2012. Shake Before Building - Replacing Make with Haskell. In *Proceedings of ICFP 2012*. ACM, 55–66.

Neil Mitchell. 2020. File access tracing. (2020). https://neilmitchell.blogspot.com/2020/05/file-tracing.html.

Andrey Mokhov. 2019. Stroll: an experimental build system. (2019). https://blogs.ncl.ac.uk/andreymokhov/stroll/.

Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build systems à la carte. *Proceedings ACM Programing Languages* 2 (2018), 1–29. DOI:http://dx.doi.org/10.1145/3236774

Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. 2016. Non-recursive Make considered harmful: build systems at scale. In *Proceedings of Haskell 2016*. ACM, 170–181.

Edmund Nightingale, Peter Chen, and Jason Flinn. 2005. Speculative execution in a distributed file system. *ACM SIGOPS operating systems review* 39, 5 (2005), 191–205.

David Patterson and John Hennessy. 2013. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

David Roundy. 2019a. Bigbro. (2019). https://github.com/droundy/bigbro.

David Roundy. 2019b. Fac build system. (2019). https://sites.science.oregonstate.edu/~roundyd/fac/.

Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 Haskell Workshop, Pittsburgh*. 1–16.

J Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd Mowry. 2000. A scalable approach to thread-level speculation. *ACM SIGARCH Computer Architecture News* 28, 2 (2000), 1–12.

J Gregory Steffan and Todd Mowry. 1998. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*. IEEE, 2–13.

Steven Swanson, Luke McDowell, Michael Swift, Susan Eggers, and Henry Levy. 2003. An Evaluation of Speculative Instruction Execution on Simultaneous Multithreaded Processors. *ACM Transactions on Computer Systems* 21, 3 (2003), 314–340. DOI:http://dx.doi.org/10.1145/859716.859720

Cristian Tapus. 2006. *Distributed speculations: providing fault-tolerance and improving performance*. Ph.D. Dissertation. California Institute of Technology.

Robert Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33.

Adam Welc, Suresh Jagannathan, and Antony Hosking. 2005. Safe futures for Java. In *Proceedings of OOPSLA 2005*. 439–453.