Riker: Always-Correct and Fast Incremental Builds from Simple Specifications

Charlie Curtsinger, *Grinnell College;* Daniel W. Barowy, *Williams College*

https://www.usenix.org/conference/atc22/presentation/curtsinger

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

RIKER: Always-Correct and Fast Incremental Builds from Simple Specifications

Charlie Curtsinger
Grinnell College

Daniel W. Barowy
Williams College

Abstract

Build systems are responsible for building software correctly and quickly. Unfortunately, traditional build tools like `make` are correct and fast only when developers precisely enumerate dependencies for every incremental build step. Forward build systems improve correctness over traditional build tools by discovering dependencies automatically, but existing forward build tools have two fundamental flaws. First, they are incorrect; existing forward build tools miss dependencies because their models of system state are incomplete. Second, they rely on users to manually specify incremental build steps, increasing the programmer burden for fast builds.

This paper introduces RIKER, a forward build system that guarantees fast, correct builds. RIKER builds are easy to specify; in many cases a single command such as `gcc *.c` suffices. From these simple specifications, RIKER automatically discovers fast incremental rebuild opportunities. RIKER models the entire POSIX filesystem—not just files, but directories, pipes, and so on. This model guarantees that every dependency is checked on every build so every output is correct.

We use RIKER to build 14 open source packages including LLVM and memcached. RIKER incurs a median overhead of 8.8% on the initial full build. On average, RIKER's incremental builds realize 94% of `make`'s incremental speedup with *no manual effort* and *no risk of errors*.

1 Introduction

Build systems specify how code and other assets should be transformed into executable software. They capture compilation procedures left unstated in the source code itself. Build systems make the process of building software more reliable, since they free programmers from having to reproduce long sequences of build commands after making a change.

To be useful, build systems should satisfy two goals. First, builds must be correct: running a build should always have the same effect, whether code was built previously or not. Second, builds must be fast: they should perform the minimum amount of work required to update a build in response to a change. These goals are often in tension. Builds that are simple to specify expose few opportunities for fast incremental updates, while complex incremental builds are more likely to be incorrect. To illustrate this challenge, we begin with an example build specification for `make`, one of the earliest and most widely-used build tools [8].

```
program①: main.c x.c x.h y.c y.h②
    gcc -o program main.c x.c y.c③
```

A `make` build specification, written in a `Makefile`, lists a collection of *build rules*. The example above shows a rule that produces a single program from three source files and two include files. Each target is composed of three parts: ① a *target name*, usually the name of an output file; ② a list of *dependencies*, which `make` calls "prerequisites," required to produce the target; and ③ a *recipe* that includes the build commands needed to produce the target from the dependencies.

At the start of each build, `make` compares the last modification time of every target with its dependencies. When a dependency is newer than the target, `make` runs the recipe to update the target. When a dependency does not exist, `make` recursively runs the rule that builds the dependency.

The key insight in `make`'s design is that developers rarely change an entire codebase between builds. Rebuilds can run faster by doing work proportional to the number of changed dependencies, not the total number of files. In the simplest case, when no dependencies change, `make` does nothing at all.

It is easy to see that the example `Makefile` is correct because all the dependencies are present and only a single build command is needed. Unfortunately, this build is also inefficient. Changing `x.c` will cause `gcc` to recompile all three `.c` files, even though `y.c` and `main.c` are unchanged. The cause of this inefficiency is that the build is monolithic: there is only one target that depends on all source files, so `make` must run a full build when any source file changes.

Monolithic builds are prohibitively expensive for larger projects. For example, a full build of LLVM takes nearly 20 minutes when run in parallel on a typical developer workstation—far too long for a developer to wait to test a small code change. Not surprisingly, large projects often have incremental build specifications. To produce an incremental `make` build, developers must break a specification into smaller rules, exposing intermediate targets. The following modifies the original `Makefile` so that it can be built incrementally.

```
program: main.o x.o y.o
  gcc -o program main.o x.o y.o
main.o: main.c x.h y.h
  gcc -c -o main.o main.c
x.o: x.c
  gcc -c -o x.o x.c
y.o: y.c y.h
  gcc -c -o y.o y.c
```

The updated specification states how to build a `.o` file from each source file, and how those `.o` files are combined into the final target, `program`. The new `Makefile` describes the same work that `gcc` performs internally; internal steps have simply been exposed so that `make` can run them or skip them during an incremental rebuild. With this `Makefile`, modifying `x.c` no longer rebuilds every output. Instead, the build generates a new `x.o`, linking it with the other `.o` files already on disk.

This `Makefile` also illustrates the dangers inherent in more complex build specifications: missing dependencies. Suppose `x.c` includes `x.h`. The original `Makefile` is correct, but the refactored one is not because changing `x.h` does not trigger a rebuild of `x.o` as it should. The implication of such a bug is that a developer with a previously-built working copy could end up with different output than another developer who starts with a clean copy of exactly the same source code.

An incremental build must be *consistent* with the full build; it should always produce output that could have come from a full build. Make and build tools like it do not guarantee this property because they do not check for missing dependencies. These errors occur with alarming frequency. A recent study showed that more than two-thirds of the open-source programs analyzed had serious build specification errors [28].

To address build errors, recent work proposes the idea of a *forward build system* [29]. Build systems like `make` are "backward" because evaluation proceeds from the final output rules, recursively building dependencies as needed. A forward build specification instead lists a sequence of commands, in order, that perform a full build. Critically, forward build systems discover dependencies automatically using program tracing instead of asking users to enumerate dependencies. On rebuild, a forward build system runs only the commands necessary to update the build, just as `make` does. When correctly designed and implemented, forward build systems guarantee correctness because they never miss dependencies.

Unfortunately, prior forward build systems miss dependencies because they fail to account for the complexity of real builds. Worse, they require users to manually specify incremental build steps. As a result, prior forward build systems are neither automatically correct nor automatically fast.

This paper introduces RIKER, a forward build system that delivers the benefits of an incremental build system with the simplicity of monolithic specifications. RIKER substantially advances the state of the art in forward build systems by using a completely different algorithmic approach. With RIKER, efficient incremental builds can be specified using a single build command like `gcc *.c`.

RIKER captures dependencies on directories, pipes, links, and sockets—not just files—ensuring that builds are correct. RIKER infers fine-grained steps from monolithic build specifications, ensuring that builds are fast. In the above example, RIKER captures the execution of the C compiler (`cc1`), assembler (`as`), and linker (`ld`), can run these commands incrementally, and in many cases in parallel, to update the build. In short, RIKER makes it possible for users to specify builds that are simple, correct, and efficient.

Contributions

Tracing and TraceIR. We present a high-performance system call tracing mechanism that generates dependence-checking programs in the novel TraceIR language (§4). TraceIR captures all of a build's state interactions—including, but not limited to, paths, files, directories, and pipes. TraceIR facilitates correct handling of circular and temporal dependencies, complexities that occur in real builds.

RIKER Build Algorithm. The RIKER algorithm performs efficient incremental builds by mixing emulation of previously-recorded TraceIR with re-execution of commands whose dependencies have changed (§5).

Implementation and Evaluation. We present an implementation of RIKER for Linux, and evaluate this implementation by using it to build 14 real-world software projects (§6). Our evaluation shows that RIKER is a significant advance over the previous state of the art in forward build tools, automatically performing incremental builds that are competitive with manually-written `make` builds. RIKER is available under an open-source license at https://rkr.sh.

2 Related Work

Build systems have evolved significantly since `make`'s introduction in 1976 [8, 21]. However, all build systems share the two goals we identify in the introduction: builds must be correct and they must be fast. Build systems differ substantially in their configuration, the precision of their dependency tracking, how changes are detected, and the level of manual effort required to use them. These differences, which we discuss below, are summarized in Table 1.

Backward Build Systems. The `make` build system and most of its successors are *backward build systems*. With a backward build system, the user writes a rule for each target produced by the build, lists the dependencies required to produce the target, and provides the commands required to create the target from its dependencies. When a target's dependency does

		Source & Build Language		Dependencies			Incremental Builds	
	Build System	Language-Agnostic	No DSL	Precise	General	Automatic	Dynamic	Automatic
Backward	Make, Ninja, Shake, Tup	✓						
	Vesta	✓		✓		$\checkmark^\star$	✓	
	CMake, Ant/Maven, SCons					$\checkmark^\lambda$		$\checkmark^\lambda$
	Pluto				$\checkmark^\star$	$\checkmark^\lambda$	✓	$\checkmark^\lambda$
	Bazel, Buck			✓	$\checkmark^\star$		$\checkmark^\star$	$\checkmark^\lambda$
Forward	Memoize, Fabricate	✓	$\checkmark^\star$	✓		✓		
	Rattle	✓		✓		✓	✓	
	RIKER	✓	✓	✓	✓	✓	✓	✓

Table 1: A comparison between RIKER and prior build tools. A build system is *language agnostic* if it can build projects written in any source language or combination of languages, and has *no DSL* if builds can be specified in a general-purpose language that is executable independent of the build system. A build system is *precise* if it ensures that a specification captures all dependencies. It is *general* if it allows cyclic dependencies, anti-dependencies, and dependencies on non-file objects. A build system is *automatic* if it discovers dependencies or incremental builds without manual specification. A tool that supports *dynamic* incremental builds can discover dependencies while a build runs. A $\checkmark^\star$ indicates partial support, and $\checkmark^\lambda$ indicates that a feature's support is language-specific.

not exist, a backward build system calls the rule that produces the dependency. Such build specifications are essentially an edge-by-edge encoding of a dependence graph.

Make, Tup [27], Shake [22], and Ninja [1] are all examples of backward build systems that require a dependence graph encoding. Writing build specifications for these tools can be burdensome. CMake [4], Ant/Maven [2, 3], SCons [26], and Pluto [5] reduce this burden by providing standard templates. Templates encode common build procedures like producing an executable from a collection of C source files. These tools automatically discover dependencies and run incremental builds, but only for supported languages. Users must provide extensions to these build tools before they can use them to build projects that use unsupported languages.

The early and innovative Vesta build system specifies builds in a general-purpose modeling language [12, 13]. Although users encode dependencies manually, Vesta uses a form of black-box tracing at the filesystem layer to identify and cache unspecified build outputs for incremental speedups. As Vesta is also a source control management tool, it can correctly reuse cached build objects in a distributed setting. Vesta can only skip work for explicitly enumerated build steps.

Buck [7] and Bazel [11] focus on ensuring that dependencies are *precise* by intentionally failing when any dependency is not explicitly provided. This is in direct contrast to Make, Tup, and Shake, where missing dependencies can lead to incorrect builds. Like CMake, Buck and Bazel simplify the process of specifying incremental builds with templates for supported languages. Buck, Bazel, and Pluto also offer some degree of dynamism when building; they can discover some additional dependencies or prune work as the build proceeds.

RIKER differs from all these tools because it precisely captures fine-grained dependencies and provides automatic speedups without manual specification, and it does so for projects written in any programming language.

Forward Build Systems. Memoize [19], Fabricate [14], and Rattle [29] use tracing to discover dependencies from a sequence of build commands that should run in order. These systems guarantee precise dependencies while remaining language agnostic. However, all of them are limited to modeling file state. A change to unsupported state, like a directory, does not trigger a rebuild, producing an incorrect rebuild. As we discuss in this work, correct builds must model not just files, but inode metadata, directories, symbolic links, hard links, pipes, and sockets. Correct build systems must also model the *absence* of such state. Existing forward build tools also have limited ability to produce fast incremental builds. Compiler drivers like `gcc` launch many separate sub-commands to compile, assemble, and link programs. Prior forward build systems require users to enumerate incremental build steps, which makes writing build specifications more difficult.

Like the above tools, RIKER is a forward build system. RIKER substantially improves the state of the art by automatically inferring fine-grained build steps. RIKER can directly invoke sub-commands called by wrapper programs like `gcc`. Sub-command execution is possible because RIKER's dependence tracking is both precise and complete.

Separation of Concerns. Many build tools tackle additional tasks, such as platform detection or compilation in a distributed setting. For example, Autoconf [9] and Automake [10] detect characteristics of the local system to generate build configurations, usually as a precursor to running `make`. Buck, Bazel, Vesta, and CloudBuild [6] offer support for distributing builds on cloud services. We regard these objectives to be orthogonal to this work. Since a RIKER build specification is just a program, it can include configuration steps. And we suspect that RIKER's precise dependency tracking may make it easier to distribute builds across machines, although we have not explored this topic.

RIKER's design is strongly influenced by the UNIX philosophy to make each program "do one thing well" [20]. Therefore, we focus this work narrowly on one problem: to build software correctly and quickly. This paper provides evidence that the key to building software correctly and quickly is the accurate detection and handling of changed dependencies, regardless of language. The fact that RIKER can be used to orchestrate compilation *for* any language, using a specification written *in* any language, is a useful property that emerges from the exclusive pursuit of this sole concern.

3 Overview

RIKER builds software using a simple specification called a `Rikerfile`. A `Rikerfile` is typically a short shell script, although it can be any executable that performs a full build. RIKER is designed to spare developers the error-prone work of specifying dependencies. Instead, RIKER discovers them automatically as a `Rikerfile` executes. RIKER uses system call tracing to capture all of a build's stateful interactions, which it records in a novel intermediate representation called TraceIR. A TraceIR *transcript* is a *program* that describes a build's dependencies, operations, and effects (see §4).

When a user requests a rebuild, RIKER evaluates the stored transcript instead of running a full build (see §5). Evaluation updates an in-memory model of system state; the model captures the effects that *would occur* if a full build were run again. Every statement in the TraceIR language is a predicate: whenever the modeled result of a TraceIR statement differs from the observed outcome of the previous build, a dependency has changed. Any command whose dependencies change must be re-executed to update the build.

During TraceIR evaluation, RIKER must decide: should a command be *executed*, or can it be *skipped*? If a command is executed, RIKER actually runs the command using the `execve` system call, tracing its execution and replacing its old TraceIR statements in the transcript. If a command is skipped, RIKER instead *emulates* the command, replaying its effects (if it has any) in the model and only *syncing* those effects to the filesystem at the end of the build or when an executed command needs to observe them. In general, emulation is orders of magnitude faster than execution. RIKER runs fast incremental builds by skipping commands whenever it can; the number of executed commands is roughly proportional to the number of changes.

RIKER repeatedly re-evaluates the entire trace until no changes are found. Re-evaluation is necessary because an executed command can change a dependency for another command. Instead of conservatively running all commands that *might* observe changes, RIKER defers the decision to execute until it can prove that a command *must* execute. Once no commands must execute, the build is "up to date" and RIKER saves the updated transcript for use in the next rebuild.

3.1 Examples

In this section, we highlight three scenarios from the working example (from §1) that illustrate RIKER's operation. The following `Rikerfile` specifies the example build:

```
#!/bin/sh
gcc -o program *.c
```

Scenario 1: Running the first build. A user runs `rkr` to perform a build. When `rkr` is invoked with no saved state RIKER starts a full build by executing `Rikerfile`.

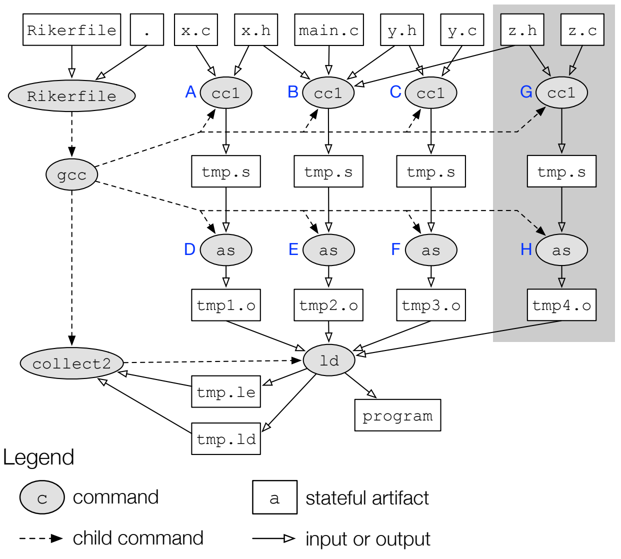

Figure 1: A dependency graph for the running example. Dashed edges show commands launching child commands, and solid edges indicate inputs and outputs. The grey box contains the modification induced by adding `z.h` and `z.c` to the build in **Scenario 2**.

Figure 1 is a dependence graph for the running example. Oval vertices represent *commands*, which correspond to programs run via `exec` system calls. Rectangular vertices represent stateful *artifacts* such as files or directories. Dashed edges indicate that a parent command *launched* a child command. Solid edges indicate a command's input or output.

In contrast to many other build systems, RIKER's build algorithm does not store build information as a dependence graph. Dependence graphs lack critical temporal information needed to disambiguate rebuild logic that arises from circular dependencies. We show Figure 1 to illustrate that even simple builds have complex dependence structure that RIKER can exploit. In fact, for clarity, Figure 1 omits a great deal: dependencies on system includes, shared libraries, the executable files for each command, and intermediate states of artifacts written multiple times during the build.

When the `Rikerfile` command launches `gcc`, it launches three instances of `cc1` in turn. Each `cc1` instance compiles a `.c` file (and any included `.h` files) to a `.s` assembly file. `gcc` also launches three instances of `as` to produce `.o` object files from each `.s` input. Finally, `gcc` launches the `collect2` command, which launches the linker `ld`. `ld` redirects `stdout` and `stderr` to temporary files which can cause `collect2` to rerun the linker with different options. Note the cycle between `collect2` and `ld`; this dependence cycle is present in every build that uses `gcc`. Also observe that `gcc` repeatedly reuses the same tmp.s temporary file, truncating it at the start of each cc1 execution. File reuse and cyclic dependencies occur frequently in builds.

Instead of a dependence graph, Riker operates over a TraceIR transcript. Riker translates every intercepted system call into a sequence of TraceIR statements. The following transcript excerpt is generated during our example build:

```
1  sh_0 = Launch(rkr , "sh Rikerfile", [...])
2  r16 = PathRef(sh_0, CWD, ".", r--)
3  ExpectResult(sh_0, r16, SUCCESS)
4  MatchMetadata(sh_0, r16,
     [uid=100, gid=100, type=dir, perms=rwxrwxr-x])
5  MatchContent(sh_0, r16, [dir: {"Rikerfile",
     "main.c", "x.c", "x.h", "y.c", "y.h"}])
```

The transcript above records that Riker launched the Rikerfile program (line 1). Linux resolved the current working directory path (line 2) without error (line 3), and the resolved directory has the given metadata and directory entries. During tracing, Riker interprets all transcribed information as what *should happen* during the normal course of a build. While Rikerfile's access of the current working directory's path resolved without error in this example, path resolution failing with ENOENT is a normal occurrence whose outcome must be recorded. For example, when the user types gcc at the prompt, UNIX may first try to access gcc in ~/bin, which will fail if ~/bin does not contain gcc. This behavior is the result of the fact that UNIX searches the user's $PATH during resolution [16]. The observed sequence of failures is a build dependency, and any change in failures implies that a build must rerun. We call failing resolutions *anti-dependencies*.

In the next section, we examine how the above transcript guides a rebuild after a user makes a code change. We defer discussion of TraceIR semantics to Section 4.3.

Scenario 2: Adding a file. Suppose a user adds files z.c and z.h and modifies main.c to include z.h. The user then runs rkr to update the build. The grey box in Figure 1 shows the effect of the change on the build's dependence graph. An efficient build system should not rebuild files unrelated to a change. Here, tmp1.o and tmp3.o do not need updating since they do not depend—even transitively—on any of the changes. At the very least, cc1 and as should be called to compile main.c and z.c, and collect2 and ld should be called to link the output to our preexisting object files. In fact, the very least is exactly what Riker does here.

Riker performs an incremental rebuild of the example by evaluating the TraceIR from the previous build. We assume the user does not change ownership or permissions for the current directory, so lines 1–4 evaluate just as before. However, line 5, which depends on directory contents, reports a change because the current directory contains the new files z.c and z.h. Riker therefore reruns and traces the Rikerfile. When the Rikerfile command is rerun, the Rikerfile's portion of the transcript is replaced with newly generated TraceIR.

Although rerunning the Rikerfile might seem to imply that the entire build will run again, this is not the case. When Rikerfile launches gcc, Riker lets the execution proceed (also under tracing) because gcc's arguments, which now include z.c, also change. However, Riker *skips* execution of the commands labeled A, C, D, and F in Figure 1, emulating them from the trace instead.

Command skipping. Let us examine the first command that Riker skips, the instance of cc1 labeled A in Figure 1:

```
1  cc1_1 = Launch(gcc_0,  "cc1 x -o tmp.s", [...])
2  r71 = PathRef(cc1_1, CWD, "x.c", r--)
3  ExpectResult(cc1_1, r71, SUCCESS)
4  MatchMetadata(cc1_1, r71,
     [uid=100, gid=100, type=file, perms=rw-rw-r--])
5  MatchContent(cc1_1, r71,
     [mtime=1619457130, hash=3c6ea, cached=false])
6  r75 = PathRef(cc1_1, r3, "tmp.s",
     -w- truncate create (rw-rw-rw-))
7  ExpectResult(cc1_1, r75, SUCCESS)
8  UpdateContent(cc1_1, r75, [hash=054521])
```

The key observation is that emulation of this command's transcript—which Riker always does before concluding that a command must run—detects no changes. cc1 reads x.c (lines 4–5) and writes to a temporary file, tmp.s (lines 6–8). The file x.c is unchanged. Although tmp.s was created by gcc and is reused by every cc1 process started by gcc, there is no dependency on that file's content because cc1 truncates the file without reading it (line 6). Because Riker detects no changes, it can emulate rather than execute cc1. Similar reasoning allows Riker to skip C, D, and F.

Scenario 3: Making an inconsequential change. Suppose a comment is added to x.c and the build is run again. This change has no effect on the final compiled program and an ideal build system should do almost nothing. Referring again to the previous trace, Riker detects a change (line 5) for cc1 because x.c changes. However, Riker correctly halts the build without ever running as, collect2, or ld.

Here is an excerpt of the as command's transcript:

```
1  as_1 = Launch(gcc_0, [as -o tmp1.o], [...])
2  r26 = PathRef(as_1, r3, "tmp1.o",
     rw- truncate create (rw-rw-rw-))
3  ExpectResult(as_1, r26, SUCCESS)
4  r27 = PathRef(as_1, r3, "tmp.s", r--)
5  ExpectResult(as_1, r27, SUCCESS)
6  MatchMetadata(as_1, r27, [uid=100,
     gid=100, type=file, perms=rw-------])
7  MatchContent(as_1, r27, [mtime=1619458806,
     hash=10732f, cached=true])
8  UpdateContent(as_1, r26, [mtime=1619458806,
     hash=3814e7, cached=true])
```

The as command reads tmp.s and writes to tmp1.o. Riker concludes that cc1's output tmp.s is unchanged (lines 6–7), even though the mtime is different. In Riker, mtimes provide a fast path for change detection: a file's content is unchanged if its mtime is unchanged, or if its content produces the expected hash. Finally, we see as writes to tmp.o (line 8). Because as is unchanged, Riker can instead restore the tmp.o file from its cache and skip as.

3.2 Summary

RIKER runs efficient incremental builds from simple specifications, even those with a single command. Throughout this section we never once needed to change the `Rikerfile`, even though the build's dependencies changed with the addition of new files; RIKER always ensures that incremental rebuilds produce the same effect as full builds. Finally, RIKER is language agnostic: it can be used to build programs written in any language using any executable build specification.

4 Tracing and TraceIR

This section defines terms that we use to describe RIKER's operation, describes RIKER's system call tracing mechanism, and introduces the novel TraceIR language that RIKER uses to record dependency information.

4.1 Definitions

An *artifact* represents system state such as a file, a directory, or a pipe. A *version* represents an artifact's state at a point in time during the build. Every artifact has both content and metadata versions. Two content versions are *identical* if and only if they contain the same bits. Two metadata versions are identical if and only if their permission bits and ownership are the same.

A *command* is a program that accesses or modifies artifacts, and may launch additional commands. A *dependency* is any version made visible to a command through a system call. A command is *changed* if its dependencies are not identical to the versions observed during the previous run, or if the outcome of an operation like path resolution differs from what was observed during the previous build (e.g. a file referenced during the last build no longer exists).

Versions are *equivalent* if they are identical, or if they could be written to the same artifact by two executions of a non-deterministic command given equivalent inputs. Outputs from different executions of a command given equivalent inputs are interchangeable, even if those outputs are not identical.

A *build specification* is any program that encodes a sequence of commands to run. A *build system* is a program that evaluates a build specification to perform a *build*. A *full build* executes all of the commands in the specification, while an *incremental build* produces the same effect while executing only a subset of commands.

When executing a command, RIKER records the dependencies and effects of each command in the TraceIR language. RIKER evaluates recorded TraceIR on subsequent builds to determine whether the command has changed. A command can be *skipped* if it is unchanged, otherwise it must be *executed*.

4.2 Tracing Implementation

RIKER executes commands under system call tracing to gather a complete set of dependencies. RIKER uses `ptrace` [17] to intercept system calls. Only calls related to filesystem interaction, file descriptor management, and process creation—75 system calls in total—matter for dependency tracking. RIKER combines `ptrace` with seccomp-BPF [25] filters to avoid tracing irrelevant system calls. Nevertheless, tracing has high overhead even with filtering. To reduce overhead, RIKER injects a shared library into each build command to intercept calls to frequently-used `libc` system call wrappers like `open` and `stat` without incurring `ptrace` overhead. This approach is inspired by RR [24], with additional support for passing system call arguments through shared memory. For a build of `memcached`—a typical C project built with `gcc`—RIKER's injected library handles 95% of system calls (`ptrace` handles the remaining 5%). For `memcached`, shared library tracing reduces RIKER's full-build overhead from 1.81s (16.9%) to just 0.84s (7.8%) with no loss of tracing precision.

The next section describes the TraceIR language. Section 4.4 shows how system calls are translated to TraceIR.

4.3 The TraceIR Language

TraceIR is an executable, linear representation of a build's behavior. RIKER generates TraceIR from system call traces, and detects changes by evaluating TraceIR against a model of the filesystem. The TraceIR language describes the dependencies that are visible to each command during the build, as well as the side effects of each command's execution.

Design considerations. The design of TraceIR is guided by three important requirements. First, tracing must be *complete*, capturing dependencies on all artifacts. A missed dependency could lead to an incorrect build. Second, TraceIR records events *serially*, even if those events come from processes that run in parallel during the build. Recording events serially imposes a temporal ordering [15] on filesystem interactions. A temporal ordering makes the relationship between an access of an artifact and its corresponding version unambiguous at any given point in a build, even when that artifact is read and written concurrently. Finally, a TraceIR program represents *a single observed path of execution* for a command. If a command's dependency changes, RIKER will execute and trace the command to discover its new behavior.

Language elements. Table 2 shows the datatypes of the TraceIR language, and Table 3 shows nearly all of the statements in the TraceIR language. We omit three low-level operations for clarity. Return types are BOOL where omitted. A TraceIR program is a sequence of statements, each associated with a command c. Statements fall into four logical groups:

Artifact accesses establish a reference to an artifact. A reference can resolve successfully or result in an error. Some accesses are side effecting (e.g. to capture the behavior of `open` with the `O_CREAT` flag).

State checks record the state observed by a command c the last time it executed. If a state check fails, c observes a change and must re-execute.

TraceIR Data Types	
BOOL, INT, STRING	Normal primitive types
[T]	A list of elements of type T
OUTCOME	Success or a POSIX error code
REF	Reference to an artifact or OUTCOME
CMD	A build command
FLAGS	Flags associated with a file access
METADATAVERSION	An artifact's metadata
CONTENTVERSION	An artifact's content
SPECIALREFID	A special artifact ID: ROOT, CWD, ...

Table 2: TraceIR data types.

State updates record the effect a command had on global state, such as writes to file content or metadata, or the creation and removal of directory entries.

Command updates record command creation, termination, and when a command waits for another to exit. These statements capture the semantics of `execve`, `exit`, `wait`, and related system calls.

4.4 Generating TraceIR Transcripts

RIKER translates each traced system call to a sequence of TraceIR statements called a *transcript*. The transcript below is generated when command c issues a successful `stat("input", &statbuf)` call.

```
1  r1 = SpecialRef(c, CWD)
2  r2 = PathRef(c, r1, "input", ---)
3  ExpectResult(c, r2, SUCCESS)
4  MatchMetadata(c, r2, [uid=100, gid=100,
                     type=file, perms=rw-r--r--])
```

Line 1 is a reference to the command's current working directory. Line 2 uses the reference to resolve the path to the file `input`. This `stat` call did not include any special flags so the fourth parameter is `---`. If the file were `open`ed rather than `stat`ed, this field would request read and/or write permission. For `lstat`, a `nofollow` flag would signal that path resolution should not follow symbolic links.

Because `stat` succeeds, line 3 records a successful outcome. If `input` is removed before a future rebuild, the observed result will be `ENOENT` and RIKER will detect a change. Finally, because c observes `input`'s metadata, line 4 records the observed metadata. If the file's metadata is modified, a rebuild will detect a change.

Importantly, TraceIR also records anti-dependencies. Supposing the `stat` call originally failed with `ENOENT`, the generated TraceIR would have had the same first two lines, but line 3 would expect `ENOENT` and line 4 would be omitted. If `input` is present on rebuild, RIKER would observe that the result is not `ENOENT` and would detect a change.

5 Build Algorithm

The RIKER algorithm, shown in Figure 2, performs two tasks: it detects changes and updates the build by running commands affected by those changes. RIKER's build algorithm is always guided by a saved transcript. RIKER's fixed-point build algorithm repeatedly evaluates the build transcript, running changed commands while checking for changes made visible to other commands. The build terminates when no new changes are found.

	TraceIR Statements
Artifact Access	PATHREF (c : CMD, b : REF, p : STRING, f : FLAGS) : REF Command c resolves path p relative to b with flags f
	SPECIALREF (c : CMD, *id* : SPECIALREFID) : REF Command c references a special artifact (e.g. /, cwd)
	FILEREF (c : CMD) : REF DIRREF (c : CMD) : REF PIPEREF (c : CMD) : REF, REF SYMLINKREF (c : CMD) : REF Command c references new anonymous file, directory, etc.
State Checks	EXPECTRESULT (c : CMD, r : REF, e : OUTCOME) Command c expects r to resolve with outcome e
	MATCHMETADATA (c : CMD, r : REF, e : METADATAVERSION) MATCHCONTENT (c : CMD, r : REF, e : CONTENTVERSION) Command c expects r to have metadata or content e
	EXITRESULT (c : CMD, *child* : CMD, n : INT) Command c expects *child* to have exit code n
State Updates	UPDATEMETADATA (c : CMD, r : REF, v : METADATAVERSION) UPDATECONTENT (c : CMD, r : REF, v : CONTENTVERSION) Command c updates r with metadata or content v
	ADDDIRENTRY (c : CMD, d : REF, e : STRING, a : REF) Command c links artifact a as entry e in directory d
	REMOVEDIRENTRY (c : CMD, d : REF, e : STRING) Command c removes entry e from directory d
Command Updates	LAUNCH (c : CMD, *cmd* : [STRING], *rs* : [REF]) : CMD Command c launches child *cmd* with inherited references *rs*
	JOIN (c : CMD, *child* : CMD) Command c waits for child *child* to exit
	EXIT (c : CMD, n : INT) Command c exits with status n

Table 3: TraceIR Statements.

Changes. TraceIR statements are predicates that express expectations about state at specific points during the build. An expectation can be in regard to the outcome of path resolution, the exit code of a child command, or the content or metadata of artifacts. Predicates are checked against the the filesystem model M. This model, a set of artifacts, records the effects of any prior TraceIR statements. The model is lazily populated with actual filesystem state; predicates that refer to artifacts that have not been modified during the build are checked against actual filesystem state. When a command's TraceIR predicate fails—that is, the expected state does not match the model M—the command is changed and must execute to update the build.

Phases. RIKER's build algorithm runs in *phases*. Each phase is an evaluation of an entire TraceIR build transcript T in the context of the model M. Trace evaluation, carried out by EVALTRACE (line 6 of DOBUILD), is repeated until the set of commands that must run, R, is empty.

```
DoBuild($trpath$)
1  $i = 1, M = \{\ \}, R = \{\ \}$
2  $T$ = LoadTrace($trpath$)
3  if $|T| == 0$ then $T$ = { Launch (rkr, Rikerfile, ...) }
4  repeat
5      $M = \{\ \}$
6      $(M, T, D, R)$ = EvalTrace($M$, $T$, $R$, false)
7      $R$ = Plan($D$, $R$)
8      $i = i + 1$
9  until $|R| == 0$
10 SyncAll ($M$)
11 if $i > 1$
12     $(\_, T, \_, \_)$ = EvalTrace($M$, $T$, $R$, true)
13 WriteTrace($T$)

EvalTrace($M, T, R, post$)
1  $T' = \text{nil}, D = \{\ \}$
2  $R_{build} = \{\ \}, R_{post} = \{\ \}$
3  for $t$ in $T$
4      $c$ = CmdOf($t$)
5      if $c \notin R$
6          $(ts, M, D, \delta_{build}, \delta_{post})$ = EvalStmt($t, M, R, D, post$)
7          if $\delta_{build}$ then $R_{build} = R_{build} \cup \{c\}$
8          if $\delta_{post}$ then $R_{post} = R_{post} \cup \{c\}$
9          $T' = T' @ ts$
10 return $(M, T', D, R_{build} \cap R_{post})$
```

Figure 2: Riker's build algorithm. DoBuild repeatedly evaluates a TraceIR build transcript T in the model M until it observes no new changes. EvalTrace evaluates a single pass through T, returning an updated model and trace, command dependence graph D, and set of commands that must run, R. `@` concatenates two lists. At the end of the `repeat-until` loop, the build is up-to-date.

Emulation vs. Execution. Riker performs incremental builds by mixing execution and emulation of build commands. Riker *emulates* a command by evaluating its TraceIR to update the in-memory model M, but does not run the command. Riker *executes* a command by running it with `exec(3)`. Riker traces the system calls of the executing command to generate new TraceIR, and evaluates this new TraceIR to keep the in-memory model M in sync with filesystem state. Whether a command is emulated or executed depends on whether it was added to R during a previous phase (see §5.1).

Invariant. Riker enforces the invariant that any command whose dependencies have changed will be executed; failing to do so results in an incorrect build. Riker emulates all other commands instead of executing them.

Executing an unchanged command preserves correctness, but doing so is costly. We avoid unnecessary command executions whenever possible because they take orders of magnitude longer than emulating a command to recreate its effects. For example, when a command's output is missing the file can be restored from a cached copy (see §5.2) instead of executing the command to recreate the file. We discuss the correctness of this approach in Section 5.5.

To avoid over-approximating R (the set of commands that will run) Riker evaluates the build transcript T repeatedly (line 4). Repeated evaluation is necessary because change detection is a dynamic problem [5]. For example, suppose T contains two commands, A and B, and that B reads one of A's outputs. Suppose A must run; does B need to run? The answer is "maybe." If A produces the same output that it produced previously, then B does not need to run. We saw this exact situation in the overview's example when adding a comment to a source file. It is safe to conservatively run B any time A runs, but Riker can potentially save work by deferring the decision to run B until after A's effects are observed.

Statement evaluation. At the heart of the build algorithm is the evaluation of TraceIR statements. The purpose of evaluating a statement is twofold: to update state and to detect changes. When a command is emulated, the only state updated is M. When a command is executed, both M and the actual filesystem are updated. The semantics of change detection depend on the specific TraceIR statement (see §4.3).

EvalStmt (line 6) evaluates a TraceIR statement. Evaluation returns one or more TraceIR statements, an updated model M, a command dependence map D (see §5.3), and two flags denoting whether the statement observed a change, either a build or post-build change (see **Post-build checks** below). The returned trace statements are used in the next build phase. When a command is emulated, its trace steps are simply echoed back. We describe command execution in §5.1. After a transcript is evaluated, Riker calls Plan (line 7), which may mark extra commands to run (see §5.3).

Filesystem and model. Riker begins each phase by initializing the model M (line 5 in DoBuild). Emulated updates are discarded after each phase because those updates are replayed in subsequent phases, whereas changes resulting from execution are written directly to the filesystem. SyncAll (line 10 of DoBuild) writes all changes in the model M to the filesystem at the end of the build. This operation ensures the outputs from commands that did not need to run are written to disk.

Post-build checks. Desirable state left behind from a previous build can look like a change to a command run in an early part of a build. Riker finishes every build with a series of *post-build checks* to avoid doing unnecessary work in this scenario. To illustrate, recall our working example from section 3 which runs the command `gcc -o program *.c`. Before performing any compilation steps, `gcc` will `stat` the `program` file, which does not yet exist. The `stat` system call produces the following TraceIR:

```
1  r8 = PathRef(gcc_0, CWD, "program", ---)
2  ExpectResult(gcc_0, r8, ENOENT)
```

Later in the build, `program` is created by the linker. As a result, immediately running the build again after completing the first full build would detect a change on line 2, and `gcc` would run. However, running `gcc` is unnecessary because the observed change is a byproduct of the build itself. Post-build checks enable RIKER to skip over these changes by encoding multiple justifications to skip: a command is unchanged if all of its predicates match what was observed during the build, or if the predicates match state found *immediately after* the conclusion of the build.

The post-build phase (line 12 of DOBUILD) emulates all commands in T, but adds additional predicates to check post-build state. After running the post-build phase in the example build, the above TraceIR excerpt is extended to capture either outcome of the `stat` call:

```
1  r8 = PathRef(gcc_0, CWD, "program", ---)
2  ExpectResult(gcc_0, r8, ENOENT) [build]
3  ExpectResult(gcc_0, r8, SUCCESS) [post-build]
```

The predicate on line 2 is the same as before, but has been marked as a *build* predicate. The new predicate on line 3 describes an alternative. With post-build checks, a command is only changed if its predicates fail in both the *build* and *post-build* scenarios (line 10 of EVALTRACE). In other words, a command is changed only when its dependencies are distinct from both the dependencies observed during the last build and immediately after the last build.

5.1 Command Emulation and Execution

RIKER begins every build by emulating a *root command*. The root command sets up initial references (e.g. the root directory, working directory, standard streams, etc.) and then launches `Rikerfile`. When an emulated command contains a LAUNCH statement, RIKER will either emulate or execute the child command depending on whether or not the child is in R. If the child is in R RIKER will launch the command in a new process with system call tracing. All statements from the child command are discarded from the transcript, and will be replaced with new TraceIR collected from the child's execution. If c is not in R, RIKER simply emulates the child from the build transcript.

Parent commands typically wait for their children to exit using the `wait` system call; this system call generates a JOIN statement in the build transcript. RIKER emulates a JOIN statement by handling traced system calls from build processes until the child command's main process exits.

When a command c is executed it may depend on artifacts modified by emulated commands. The latest state of these artifacts is in M, not on the actual filesystem. RIKER will write these changes out to the filesystem as they are needed, either when c begins execution (when file descriptors are inherited by the child) or when c first accesses the artifact during its execution. This mechanism is critical for RIKER's ability to incrementalize builds, and relies heavily on caching.

5.2 Caching

Without caching, RIKER's ability to execute sub-commands in isolation would be limited because many of the needed inputs would not be available. `gcc` and other language tools routinely create "ephemeral" state—like temporary files—as communication channels for tools in the toolchain. RIKER's caching and TraceIR make it possible to automatically restore this ephemeral state to run a command whose inputs are produced by other commands (e.g. the assembler or linker).

The motivating example in Figure 1 illustrates this functionality. Suppose a user edits `main.c`; RIKER will execute the compiler and assembler to produce `tmp2.o`, but does not need to execute any commands to produce the other `.o` files. Instead, these files are simply restored from cache when they are first accessed by the linker. The fact that `gcc` reuses `tmp.s` is not a problem, as each use of the file is ordered in the build transcript so RIKER always knows which version of the file is required for every command.

RIKER caches files, symlinks, and directories. Cached artifacts are stored in a `.rkr` directory, and are garbage collected when RIKER detects that they are no longer referenced by the build transcript. RIKER currently does not cache pipes, sockets, or special files. If a command that reads from a pipe must run, RIKER will also run commands that write to that pipe to provide uncached inputs.

5.3 Build Planning

The purpose of build planning (line 7 in DOBUILD of Figure 2) is twofold: to ensure the build terminates, and to improve efficiency. PLAN works much like the mark-sweep garbage collection algorithm [18] and uses the command dependence graph, D, returned by EVALTRACE. D is a digraph of producer-consumer relationships between commands.

Commands are marked under a few conditions: a) a command is in R, having directly observed a change in EVALTRACE; b) a command consumes uncached input produced by a command already marked to run; c) a command produces uncached output consumed by another command already marked to run; or d) a command produces uncached output that should persist after the build.

The above criteria identify commands that subsequent build phases in a cycle-free build would eventually identify, so for those builds it reduces the number of phases. However, in builds with dependence cycles, RIKER may not terminate without special handling. In D, a cycle appears as a strongly-connected component (SCC) [31]. Planning ensures that commands in a cycle run atomically. Caching also breaks cycles; RIKER only needs to atomically run SCCs that interact through uncached artifacts like pipes.

5.4 Exit Code Handling

Unlike prior forward build systems, RIKER can execute a sub-command without executing its parent. Parent commands can observe the exit codes of their children, so if a child finishes with a different exit code the parent must run. Because exit code changes are rare, RIKER optimistically assumes that the child's exit code will not change. In the common case the child command runs, finishes with the expected exit code, and the build is complete. If the child finishes with a different exit code, the parent observes a change and will re-execute in the next phase of the build. Executing the parent may re-execute the child if the child's dependencies change again. In the worst case, RIKER could backtrack on every command, taking $O(n^2)$ time, where n is the number of commands. Even for builds that contain compilation errors—and thus changed exit codes—we observed that total work done is close to $O(n)$.

5.5 Correctness

Here we provide a proof sketch for the correctness of RIKER's incremental build algorithm. We make the following assumptions.

A1. The user-provided full build specification does what the user intends.

A2. The user accepts that equivalent outputs (defined in §4.1) are interchangeable, an assumption shared by most build systems.

A3. Intercepting system calls is sufficient to determine all dependencies.

A4. Our translation from system calls to TraceIR faithfully captures dependencies and side-effects. Our empirical evaluation in Section 6 provides evidence that this translation is accurate.

An incremental build tool that produces outputs that could not have come from a full build is clearly incorrect. A *consistent* build produces output that could have come from a full build, and is therefore correct (A1, A2) [12, 13].

Running an empty build specification produces no output, so all rebuilds of this specification are by definition consistent. Given a consistent build with k commands, add command $k+1$ to the specification. Command $k+1$ may depend on outputs from any of the preceding k commands. The build remains consistent when $k+1$ is executed (A1). On rebuild, if $k+1$ is unchanged, skipping command $k+1$ preserves consistency because RIKER restores cached artifacts (A2). By induction, a build is consistent as long as all changed commands are executed.

RIKER executes changed commands in phases. If an executed command produces a different output read by another command, the latter command is changed and will execute in the next phase. The build terminates when a phase finishes with no changed commands. Since RIKER's algorithm executes all changed commands, it produces consistent builds.

The proof sketch above assumes commands k and j do not participate in a dependence cycle, but these cycles arise in real builds. Without loss of generality, assume k writes output that j accesses, and later j writes output that k accesses; RIKER's build transcript captures the temporal order of these interactions. We allow for such cycles by logically partitioning k into k', the portion of k that runs before its dependence on j, and k'', the remainder of k. Now j depends on k' and k'' depends on j, so there is no longer a cycle. We add the constraint that if either k' or k'' must run, both will run, as these are actually two parts of the same command.

6 Evaluation

Our evaluation of RIKER addresses four key questions:

Q1: Are RIKER builds easy to specify?

Q2: Are RIKER builds fast?

Q3: Are RIKER builds correct?

Q4: How does RIKER compare to RATTLE?

We use RIKER to build 14 software packages, including large projects like LLVM, memcached, redis, and protobuf. Evaluation was conducted on a typical developer workstation with an Intel Core i5-7600 processor, 8GB of RAM, and an SSD running Ubuntu 20.04 with kernel version 5.4.0-80. Builds use either `gcc` version 9.3.0 or `clang` 10.0.0.

6.1 Are RIKER builds easy to specify?

To answer this question, we wrote `Rikerfiles` for seven applications: lua, memcached, redis, `rkr`, sqlite, vim, and xz. The new builds produce the same targets as the projects' existing `make` or `cmake` builds. Unlike the default build systems, the RIKER-based builds do not list any dependencies or incremental build steps. Three of these builds were written by undergraduate students over the course of a few days; the students were new to RIKER and unfamiliar with the project sources they were building. The biggest challenge the students faced was understanding the existing build specifications, a task that is likely easier for the project's own developers.

A key feature of a `Rikerfile` is its brevity, illustrated by memcached's complete `Rikerfile`:

```
CFLAGS="..."
DEBUG_CFLAGS="..."
MEMCACHED_SRC="memcached.c hash.c ..."
TESTAPP_SRC="testapp.c util.c ..."
gcc $CFLAGS -o memcached $MEMCACHED_SRC -levent
gcc $DEBUG_CFLAGS -o memcached-debug \
  $MEMCACHED_SRC -levent
gcc $CFLAGS -o sizes sizes.c -levent
gcc $CFLAGS -o testapp $TESTAPP_SRC -levent
gcc $CFLAGS -o timedrun timedrun.c -levent
```

This level of simplification is typical; Our largest `Rikerfile`—used to build sqlite—is just over 5KB, and consists mostly of a list of source files. Forward builds are easier to specify because of automatic dependency discovery. RIKER's incremental builds are also significantly shorter than specifications for prior forward build tools (see §6.4).

6.2 Are RIKER builds fast?

The first full build of any software project is usually the longest build. Full builds are where RIKER incurs the largest absolute overhead. Importantly, full builds are not the common case; developers run incremental builds far more often than full builds. This section shows that even with the extra delay, full builds are reasonably fast with RIKER.

To measure RIKER's overhead, we built 14 software projects with RIKER. Seven of these projects use a `Rikerfile` that replaces the default build (see §6.1), while

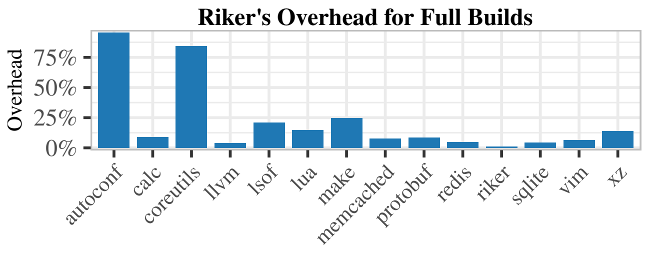

Figure 3: RIKER runtime overhead for full builds compared to each project's default build system. RIKER's median overhead on full builds is 8.8%, with a median absolute slowdown of 1.2s.

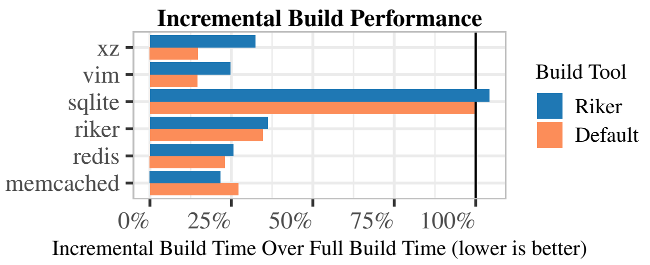

Figure 4: Time to run 100 incremental builds as a percentage of the time to run a full build at every commit using the project's default build system. Excluding sqlite, RIKER's incremental builds save 235 minutes compared to 250 minutes saved by the default build system.

the other half use a `Rikerfile` that *wraps* the default build. The only requirement for a `Rikerfile` is that it run a full build, so a `make`-based project can be built with a one-line `Rikerfile`: `make --always-make`. Unfortunately, tracing `make` itself can lead to spurious dependencies so this approach is only suitable for evaluating full-build performance.

Figure 3 shows the results of running full builds with RIKER. These builds are run in serial; we examine the performance of parallel builds later in this section. Each project is built five times with RIKER and its default build system, with the exception of LLVM which we build three times due to its long build time. Median full-build overhead for all benchmarks is just 8.8%; most builds have between 4% and 20% overhead. TraceIR transcript sizes are roughly proportional to build time, ranging from 2MB for autoconf (1.2s build) to 264MB for LLVM (77-minute build). In absolute terms, RIKER spends a median of just 1.2 seconds longer to perform a full build than each project's default build system. The longest additional waiting time for a RIKER build is for LLVM, which takes about three minutes longer than the default 77 minute build (4% overhead). The worst overheads appear in projects like autoconf and coreutils that build many small programs rather than a single large executable. RIKER's full-build overhead is less than 25% for all other projects.

Incremental Builds. The most important measure of efficiency for a build system is its ability to perform fast incremental rebuilds. We perform two experiments to measure the efficiency of RIKER's incremental builds.

First, we measure the time it takes RIKER to perform a *no-op build*—one where no commands need to run—by running an incremental build immediately after finishing a full build. The median RIKER no-op build time over the 14 benchmarks is just 220ms, compared to 5ms for the default build system. The longest additional wait is for the LLVM build, which takes 11.3s with RIKER compared to 4.8s with `make`. More than half of the no-op builds with RIKER take just 162ms longer than the default build system, an imperceptible difference.

Second, we use real developer commits to measure the efficiency of performing incremental builds with RIKER versus a project's default build system. We run this experiment on six projects—memcached, redis, `rkr`, sqlite, vim, and xz—all of which have custom `Rikerfile`s and public version control histories. We perform a full build of each project, and then measure the time required to update the build at each of the next 100 commits in the project's `git` repository. This experiment simulates a developer performing incremental builds after editing a subset of the project's source files.

Figure 4 shows the results of this second experiment. The graph shows the total time required for all 100 incremental builds as a percentage of the time it would take to run a full build at each commit. Note that we compare RIKER's incremental build times to the time it would take to run a full build with the project's *default* build system. This ensures RIKER's overhead on the full build does not give it an advantage compared to the slightly faster default build system.

95% of RIKER's incremental builds complete within 3.8s of the default build system. In every benchmark except one (sqlite), RIKER is able to reduce build time by at least 63% relative to a full build. Over 5000 incremental builds of these five benchmarks, the default build system reduces build time by 74.7%, saving 250 minutes compared to full builds at each commit. RIKER saves 70.1% of total build time—a total of 235 minutes. Building with RIKER yields 94% of the benefit of a manually-specified incremental build, but with *no manual effort* and *no risk of errors*.

RIKER is able to save more time than the default build system for memcached because the case study includes several commits that edit the build specification itself. These edits generally require a full build for `make`-based projects, but RIKER is still able to perform a safe incremental build when the specification changes.

Neither RIKER nor `make` saves any work when building sqlite. This is because sqlite's build concatenates all of its source files together before compiling them so the project can be distributed as a single source file. RIKER's overhead adds less than three seconds (4.2%) to each "incremental" build.

Parallel Builds. Our evaluation has so far focused on serial builds, but parallel builds are also a useful mechanism for reducing build times. There are two concerns when it comes to RIKER's support for parallel builds. First is scalability: how does RIKER's tracing impact the performance of parallel builds? The second concern is expressiveness: how do users write a `Rikerfile` that describes a parallel build?

To assess RIKER's scalability we focus on LLVM, our largest benchmark. On our evaluation machine, which has four cores, we see good scalability when adding up to four parallel jobs to LLVM's `make` build. Build time is reduced by 54.3%, 64.4%, and 71.9% when building with two, three, and four workers respectively. With RIKER, we see reductions in build time of 47.7%, 63.0%, and 67.0% for the same numbers of workers. RIKER's overhead increases from 4.0% for the serial build to 22.2% with four workers. A likely cause for this reduction in scalability is that RIKER utilizes a busy-wait loop that monopolizes a core. Even with this limitation—which we plan to address—parallel builds with RIKER are still significantly faster than serial builds.

While the prior experiment examines RIKER's tracing performance on a parallel `make` build, it is also possible to write parallel build specifications directly in a `Rikerfile`. To make this as easy as possible, RIKER includes a wrapper around common C/C++ compilers that launches sub-commands for compilation in parallel, which is always safe (two `.c` files can always be compiled simultaneously). With RIKER's compiler wrapper, simple lines like `gcc *.c` start parallel compilation tasks for each source file, which RIKER can also run in parallel for later rebuilds. Enabling this wrapper reduces build time for memcached by about 50%, compared to 60% with memcached's own parallel `make` build. RIKER's compiler wrapper provides an easy, automatic way to run parallel builds, and RIKER's tracing does not significantly limit scalability.

6.3 Are RIKER builds correct?

We run each project's full test suite for both the default and RIKER builds. For the six projects in the previous section, the tested outputs are the product of one full build and 100 incremental builds, one for each commit. The remaining projects run only full builds. Every RIKER project passes exactly the same tests as the original build system. This experiment provides evidence that RIKER correctly updates builds to produce equivalent final targets.

We have additional confidence that RIKER's translation from system calls to TraceIR and its POSIX filesystem model are correct because RIKER checks the outcomes of operations in the model against actual system call results. RIKER raises a warning if the model deviates from actual system behavior; our experiments and test suite raise no such warnings.

6.4 How does RIKER compare to RATTLE?

To compare against the prior state of the art forward build system, we ported the memcached build to RATTLE [23]. Our efforts to port other benchmarks to RATTLE were not successful. RATTLE limits state modeling to files, meaning RATTLE misses some kinds of changes [29, 30]. We demonstrate with the following RATTLE build:

```
main :: IO ()
main = rattleRun rattleOptions $ do
  cmd "gcc prog.c"
  cmd "mkdir dir"
  cmd "mv a.out dir"
```

The full build runs correctly. However, because RATTLE does not model directories, changing `prog.c` leads to an inconsistent rebuild. A new `a.out` is placed in the current directory, leaving the original in `dir` untouched. RATTLE cannot build sqlite for precisely this reason. RATTLE also does not handle circular dependencies. The lua build calls `ranlib`, which is used to create library archives. Since `ranlib` modifies its input, RATTLE fails during the full build.

We were able to build memcached with RATTLE, which imposes a median overhead of less than 1% for the full build. This is because RATTLE uses library interposition for tracing, which is faster than `ptrace` but will miss system calls not issued by libc. A straightforward translation of the build specification from RIKER to RATTLE does *not* result in good incremental build performance; RATTLE cannot run fine-grained incremental builds from simple specifications, so it only reduces build time by 25% compared to full builds over the 100 commits from our earlier experiment. This is significantly less than the 78% build time reduction RIKER is able to achieve from the simple build specification. A RATTLE specification with comparable incremental build performance requires 63 separate commands, compared to just five for RIKER.

7 Conclusion

RIKER significantly lowers the burden of correctly specifying fast incremental builds. A RIKER build specification can be any executable program, like a simple shell script that performs a full build. In many cases, even a single build command such as `gcc *.c` is sufficient. Users do not need to list dependencies in their build specifications, nor are they required to break builds into incremental steps. Nevertheless, RIKER always builds correctly and quickly.

RIKER uses low-overhead system call tracing to automatically discover dependencies as build commands execute, and on rebuild, runs only the subset of commands required to bring the build up-to-date. RIKER has a median overhead of 8.8% across 14 real software packages, and it realizes 94% of `make`'s incremental build speedup with no manual effort and no risk of errors. We think these substantial engineering improvements are well worth RIKER's modest overheads. RIKER is available under an open-source license at https://rkr.sh.

Acknowledgments

This work was supported by National Science Foundation grants CNS-2008487 and CNS-2008940. We thank Michael Chovanak, Yolanda Jiang, Alissa Johnson, Philip Ma, Abigail Munsen, Jonathan Sadun, and Linh Tang, who contributed to RIKER's implementation while they were students at Grinnell College. We also thank Emery Berger, John Vilk, and Benjamin Zorn for their feedback and guidance.

References

[1] The Ninja Build System. https://ninja-build.org/, November 2020. v1.11.0.

[2] Apache Software Foundation. Apache Ant. https://ant.apache.org, October 2021. v1.10.12.

[3] Apache Software Foundation. Apache Maven. https://maven.apache.org/, March 2022. v3.8.5.

[4] CMake Project. CMake. https://cmake.org, May 2022. v3.23.2.

[5] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. A Sound and Optimal Incremental Build System with Dynamic Dependencies. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 89–106, New York, NY, USA, 2015. ACM.

[6] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. CloudBuild: Microsoft's Distributed and Caching Build Service. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, page 11–20, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Facebook. Buck: A high-performance build tool. https://buck.build/, January 2021. v2022.05.05.01.

[8] Stuart I. Feldman. Make — a program for maintaining computer programs. *Software: Practice and Experience*, 9(4):255–265, 1979.

[9] GNU Project. GNU Autoconf. https://www.gnu.org/software/autoconf/, January 2021. v2.71.

[10] GNU Project. GNU Automake. https://www.gnu.org/software/automake/, October 2021. v1.16.5.

[11] Google. Bazel. https://bazel.build/, June 2022. v5.2.0.

[12] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. SRC Technical Note 1999-001: The Vesta Approach to Software Configuration Management. *Compaq Systems Research Center*, June 1999.

[13] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. *Software Configuration Management System Using Vesta*. Springer Science & Business Media, 2006.

[14] Ben Hoyt and Simon Alford. Fabricate. https://github.com/brushtechnology/fabricate, October 2020.

[15] Leslie Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System*, page 179–196. Concurrency: The Works of Leslie Lamport. Association for Computing Machinery, New York, NY, USA, 2019.

[16] Linux man-pages project. *path_resolution(7), Linux User's Manual*, November 2017.

[17] Linux man-pages project. *ptrace(2), Linux User's Manual*, March 2021.

[18] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[19] Bill McCloskey. Memoize: A replacement for make. http://www.eecs.berkeley.edu/~billm/memoize.html, June 2008. Archived: 2010-09-05.

[20] M.D. McIlroy, E. N. Pinson, and B. A. Tague. Unix Time-Sharing System: Forward. *The Bell System Technical Journal*, 57(6):1899–1904, 1978.

[21] Peter Miller. Recursive Make Considered Harmful. *Journal of AUUG Inc.*, 19(1), March 1997.

[22] Neil Mitchell. Shake Before Building: Replacing Make with Haskell. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 55–66, New York, NY, USA, 2012. ACM.

[23] Neil Mitchell. Rattle. https://github.com/ndmitchell/rattle, May 2020. v0.2.

[24] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering Record and Replay for Deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 377–389, Santa Clara, CA, July 2017. USENIX Association.

[25] The Linux Kernel Project. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[26] SCons Foundation. Scons. https://scons.org, February 2022. v4.3.0.

[27] Mike Shal. Tup Build System. https://gittup.org/tup/, May 2021. v0.7.11.

[28] Thodoris Sotiropoulos, Stefanos Chaliasos, Dimitris Mitropoulos, and Diomidis Spinellis. A Model for Detecting Faults in Build Specifications. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), November 2020.

[29] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. Build Scripts with Perfect Dependencies. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), November 2020.

[30] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. Forward Build Systems, Formally. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 130–142, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

A Artifact Appendix

Abstract

The RIKER artifact includes a virtual machine image with a pre-installed copy of RIKER. RIKER is a build system that automatically discovers and runs incremental builds from simple specifications. This artifact makes it easy to reproduce the experiments described in the paper, which evaluate RIKER's performance and effectiveness as a build tool.

Scope

This artifact should be used for two purposes: a) to reproduce the experiments from the paper, and b) to generate plots from the new results. The provided scripts reproduce the full-build overhead plot (Figure 3), the incremental savings plot (Figure 4), and the summary statistics reported in the abstract and in Section 6. We expect runtime overhead numbers will vary across platforms. Running the evaluation within a virtual machine will also likely have some effect on overhead. However, the artifact's overhead should be close to the paper's reported 8.8% median overhead on full builds, and incremental savings should be close to `make`'s incremental build performance.

Contents

README: A detailed guide to setting up the artifact and running experiments from the paper.

Virtual Machine Image: A VM image in OVA format that contains RIKER's source code, build dependencies, and scripts that automatically run the paper's benchmarks.

Hosting

The artifact is available at https://doi.org/10.5281/zenodo.6544966. Updated versions of RIKER will be available at https://rkr.sh. We recommend using the updated version for uses other than reproducing experimental results from the paper. Newer versions are likely to be more stable, support more kernel versions and architectures, and include bug fixes and additional features.

Requirements

Hardware Requirements. The virtual machine included with this artifact requires an x86_64 machine.

Software Requirements. The virtual machine image includes all build and evaluation dependencies. The OVA file can be imported into any hypervisor that supports OVA, but it was developed and tested using VirtualBox. The experimental evaluation requires network access; other hypervisors may require changes to network configuration after import.