Program Synthesis for Systems Developers

Jacob Van Geffen

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2023

Reading Commitee:

Emina Torlak, Chair

Xi Wang

Zachary Tatlock

Program Authorized to Offer Degree:
Computer Science & Engineering

© Copyright 2023

Jacob Van Geffen

University of Washington

Abstract

Program Synthesis for Systems Developers

Jacob Van Geffen

Chair of the Supervisory Committee:
Emina Torlak
Paul G. Allen School of Computer Science & Engineering

Implementing and verifying the correctness of systems software poses a difficult challenge for developers. Systems software operates across multiple levels of abstraction, requiring developers to reason about the interactions between these abstraction layers. At the same time, ensuring correctness of these systems is now more important than ever. Linux kernel vulnerabilities can allow malicious users to gain root access in critical systems, and incorrectly implemented cloud storage systems can harm data availability for millions of users.

This dissertation presents two novel *program synthesis* tools that automate the implementation and verification of two classes of systems: in-kernel just-in-time (JIT) compilers and crash consistent storage systems. The first of these tools, JitSynth, allows kernel developers to automatically generate correct in-kernel JIT compilers by giving a specification of the source and target language. These JITs translate user-submitted programs to lower-level assembly code for kernel execution. Manually implementing (and proving correctness of) these JITs poses a difficult challenge for developers due to subtle differences in the semantics of the source and target languages. By synthesizing JITs automatically, JitSynth allows developers to avoid kernel-breaking bugs without the massive effort of implementing and verifying a new compiler for each target architecture.

The second tool presented, DepSynth, enables storage system developers to automatically add crash consistency mechanisms to their systems. Designing crash consistent systems is difficult for developers because it requires reasoning about complex constraints on the orderings of storage system writes. DepSynth allows developers to reap the data availability and resiliency benefits of crash consistency without the overhead of manually reasoning about these orderings. Together, these tools demonstrate the effectiveness of program synthesis for developing systems software.

Contents

1 Introduction **1**

1.1 Synthesis for In-Kernel JIT Compilers 3

1.2 Automatic Crash Consistency with Synthesis 4

1.3 Contributions 5

2 JitSynth: Synthesizing Just-In-Time Compilers for In-Kernel DSLs **7**

2.1 Overview 8

2.2 JitSynth in a nutshell 11

2.3 Problem Statement 17

2.4 Solving the Mini Compiler Synthesis Problem 22

2.4.1 Generating Compiler Metasketches 22

2.4.2 Generating Naive Sketches 23

2.4.3 Generating Read-Write Sketches 24

2.4.4 Generating Pre-Load Sketches 27

2.4.5 Solving Compiler Metasketches 29

2.5 Implementation 30

2.6 Evaluation 31

2.6.1 Synthesizing compilers for real-world source-target pairs 32

2.6.2 Effectiveness of sketch optimizations 35

2.7 Related Work 35

2.8 Conclusion 37

3 DepSynth: Automatically Developing Crash Consistency Mechanisms 39

3.1 Overview 40

3.2 DepSynth by Example 43

3.3 Reasoning About Crash Consistency 48

3.3.1 Disk Model and Dependency Rules 49

3.3.2 Storage Systems and Litmus Tests 51

3.3.3 Reasoning About Crashes 53

3.4 Dependency Rule Synthesis 57

3.4.1 Problem Statement 57

3.4.2 The DepSynth Algorithm 58

3.4.3 Synthesizing Dependency Rules with Happens-Before Graphs 60

3.4.4 Resolving Cycles in Dependency Rules 68

3.5 Optimizations and Design Choices 69

3.5.1 Attempting graph search with a $\mathcal{T}_{ord}$ solver. 70

3.5.2 Faster Consistency Checks for Litmus Tests 71

3.6 Evaluation 73

3.6.1 ShardStore Case Study 73

3.6.2 Crash-Consistency Bugs 77

3.6.3 Other Case Studies 79

3.7 Related Work 80

3.8 Conclusion 82

4 Conclusion 83

Bibliography 85

Acknowledgments

Over the past five years, I've had the privilege of working with of Emina Torlak. She is everything I could have possibly asked for in a mentor. From getting into the Rosette weeds to career chats over Molly Moon's, she always seems to know exactly what I need to hear and when I need to hear it. Thanks for always believing me and keeping me moving forward.

Before I ever started my PhD at UW, I saw James Bornholt give a talk at PLDI 2017 about his work synthesizing memory models. Of course I loved the talk, but at the time, I had no idea how much of an impact James' work would have on my life. Without the groundwork of James' ridiculously prolific career, this thesis would not exist. As if this wasn't enough, James has also been an excellent mentor and friend throughout my time at UW. Thanks for working with me, James.

My path to writing this thesis feels like a series of insanely lucky happenstances. Starting my undergraduate degree at UT Austin the same semester that Isil Dillig became a faculty member at UT wasn't the first or last bit of this luck, but it certainly was one of the most impactful. Isil is the reason that I started my program synthesis research journey, and I likely never would have pursued a PhD without her guidance.

I'm extremely grateful to everyone that I've worked with during my time at UW. Without fail, Xi Wang always asks the most important and challenging questions, and I wouldn't be half the researcher I am today without him. Luke Nelson brought me up to speed when I started in the

UNSAT group at UW, and for that I'll always be thankful. Additionally, I'd like to thank Zach Tatlock and Shane Steinert-Threlkeld for agreeing to be part of my thesis committee. Zach, thanks for making me feel at home in new, scary surroundings.

I doubt I could have finished this PhD with my sanity intact without the support of all my friends from the UNSAT group, PLSE lab, Systems lab, and Architecture group. There are so many amazing people that have welcomed me into these communities, but especially to Oliver Flatt, Ben Kushigian, Katie Lim, Ellis Michael, Samantha Miller, Chandrakana Nandi, Pratyush Patel, Dan Petrisko, Sorawee Porncharoenwase, Max Ruttenberg, Gus Smith, Ewin Tang, and Max Willsey, thank you for making these past five years a blast.

Finally, I cannot emphasize how crucial the love and support from my family has been over the past five years. My parents Sheila and Steve and my brothers Daniel and Sammy have been a non-stop source of encouragement. Madeline, I'll never be able to thank you enough for sticking through this whole journey with me, but I will certainly try. Thank you for always being on my team.

Chapter 1

Introduction

In recent years, the development of software systems has become increasingly complex. From the construction of large production key-value stores to Linux kernel JIT compilers, modern systems must satisfy ever-increasing requirements on performance. Additonally, the correctness of these systems is now more important than ever. The afformentioned storage systems are responsible for safely storing the data of millions of people, while the incorrect behavior of in-kernel compilers can subject the tens of millions of Linux users to a slew of vulnerabilities.

Like software development for any domain, systems software development involves three fundamental tasks: **specifying** correct behavior at a high level, **implementing** the system at a low level, and **verifying** that the implementation correctly adheres to the specification. Where systems software distinguishes itself is in the level of abstraction at which the implementation operates. Specifically, many systems must operate between layers of abstraction, such as between source and target languages or between the application and hardware layers of the operating system. This means that manually implementing and verifying systems software requires a high amount of effort from the developer, who must reason about the multiple abstraction layers in addition to the execution of their implemented system. Additionally, stringent requirements on

performance of systems software further increase the difficulty of implementation and verification.

A well-studied technique for reducing the burden of software implementation and verification is *program synthesis*. Program synthesis is an automated programming technique that takes advantage of overlapping semantics between the three tasks described above in order to automate software implementation and verification. To do so, synthesis tools take as input a specification and output both an implementation as well as a certificate verifying that the implementation correctly behaves according to the input specification.

The goal of my work is to automate system development with program synthesis. Systems software is a particularly appealing target for program synthesis due to the difficultly of manually implementing high-performance systems that operate accross multiple abstraction layers. However, for the same reasons that developers struggle to manually reason about systems, creating tools to *automatically* construct systems software poses a uniquely difficult challenge. Synthesis tools for systems must not only consider the space of candidate implementations, but also the complex semantics of the abstractions that the system implementation operates between. For JIT compiler synthesis tools, this means considering all possible executions of both source and target programs; for crash consistency automation, this means reasoning about sets of disk writes that a given storage system may issue, as well as any possible reorderings of those writes made by the disk hardware. Moreover, demands on the performance of systems further complicate this task, restricting the space of acceptable outputs from a systems synthesis tool.

To overcome these challenges, this dissertation presents novel *system decomposition* and *metaprogramming abstraction* techniques. **System decomposition** allows synthesis tools to generate smaller target programs. By considering systems as a conjunction of smaller pieces, synthesis tools can search for candidate programs that implement a single piece. The resulting synthesized program can be combined with either user-written code or other synthesized system pieces, effectively reducing the developer burden. Second, system-specific **metaprogram**

abstractions enable synthesis tools to reason about candidate programs that themselves take DSL programs as input. By reasoning about these simpler abstractions rather than the complex systems directly, synthesis tools are able to more efficiently search over candidate programs. In the next two sections, I describe how these observations enabled the creation of synthesis tools for both in-kernel JIT compliers and for crash consistent storage systems.

My thesis is that *program synthesis is an effective tool for designing systems that are both correct and performant.* This dissertation explores this thesis over two particular types of systems — just-in-time compilers and crash consistent storage systems — and demonstrates how new synthesis tools can reduce implementation and verification effort for developers despite the size and complexity of these systems.

1.1 Synthesis for In-Kernel JIT Compilers

Modern operating systems have become increasingly extensible and customizable. One way these systems enable customization is by allowing users to submit custom code for kernel execution written in an *in-kernel DSL*. Initially designed for network packet filtering, in-kernel DSLs like eBPF [Fle17] enable kernel extensions such as performance monitoring, load balancing, and intrusion detection [Eng96, Fle17, MJ93]. For performance reasons, operating systems use just-in-time (JIT) compilers to execute in-kernel DSL programs. However, these JIT compilers can be the source of major kernel bugs [Hor18, Pau20, Bla10, KHF+19]. Moreover, writing JIT compilers can pose a substantial effort to developers, requiring on the order of thousands of lines of code.

Chapter 2 presents JitSynth, a program synthesis tool that automatically generates JIT compilers for in-kernel DSLs given a specification of the source and target language. Existing work has demonstrated how the correctness of JIT compilers can be proven manually[WLZ+14, Sob15], but the effort of such proofs is non-trivial. By synthesizing JIT compilers, developers both avoid the

burden of implementing such systems *and* can guarantee correctness without requiring intensive proofs.

In order to synthesize such massive and complex systems, JITSYNTH considers the synthesis problem as one over per-instruction compilers between *abstract register machines* — an abstraction over in-kernel DSLs and other low-level languages introduced in Chapter 2. JITSYNTH decomposes the synthesis problem for per-instruction compilers into multiple synthesis tasks for *minicompilers* that translate only one source instruction. To prune and prioritize the search space for minicompilers, JITSYNTH takes advantage of a novel *compiler metasketch* that enables the efficient exploration of candidate minicompilers using off-the-shelf satisfiability solvers. With JITSYNTH, we have synthesized three JIT compilers: one from eBPF to RISC-V, one from classic BPF to eBPF, and one from libseccomp to eBPF. Moreover, we have confirmed that the synthesized eBPF to RISC-V compiler lacks bugs previously found in the existing Linux compiler.

1.2 Automatic Crash Consistency with Synthesis

Storage systems face an increasing demand for reliability. From Linux file systems to large cloud-based key-value stores, storage systems must maintain the integrity of their data even in the presence of system crashes. This property of storage systems is known as *crash consistency*. Guaranteeing crash consistency is a challenging task for developers, as it requires maintaining complex invariants that depend on storage system implementation details. Moreover, failing to guarantee crash consistency may result in severe impacts, including costly slowdowns and massive data losses.

Chapter 3 presents DEPSYNTH, a tool for automatically generating crash consistency code for storage systems. DEPSYNTH takes advantage of the observation (identified in [FMK+07]) that crash consistency mechanisms fundamentally enforce *write-before relationships*. To automate crash

consistency, DepSynth takes as input the implementation of a (non-crash consistent) storage system together with a set of example programs over the system called *litmus tests*. The output is a set of *dependency rules* which specify, at runtime, the required write-before relationships to enforce.

We take advantage of several novel abstractions in order enable automatic crash consistency with DepSynth. First, we assume a new *angelic crash consistency* model that delegates enforcement of write-before relationships to a *dependency-aware buffer cache*, which sits between the storage system software and disk hardware layers. Second, we introduce a language for dependency rules that allows for both *expressive* and *generalizable* rules. Finally, we describe a new search for dependency rules directed by input litmus tests. These insights enable DepSynth to synthesize dependency rules for an existing production key-value store at Amazon [BJA+21] in 49 minutes minutes.

1.3 Contributions

The remainder of this dissertation is broken into two chapters. Chapter 2 introduces JitSynth and the techniques that enable JitSynth to synthesize in-kernel JIT compilers. By decomposing the search with minicompilers and efficiently exploring the space with novel metasketches, JitSynth is able to synthesize correct JITs whose performance nears that of their hand-written counterparts.

Chapter 3 presents DepSynth, a tool for automating crash consistency in storage systems. In Chapter 3, we introduce a new dependency rule language that describes rules which specify required write-before relationships for a storage system at runtime. This abstraction over write-before relationships, along with a new per-litmus-test search, allows DepSynth to synthesize correct dependency rules for a production system. Moreover, the generated rules have nearly the same effect on performance as dependencies hand-written by experts. These contributions

demonstrate how program synthesis can aid developers in building correct and performant systems.

Chapter 2

JitSynth: Synthesizing Just-In-Time Compilers for In-Kernel DSLs

Modern operating systems allow user-space applications to submit code for kernel execution through the use of in-kernel domain specific languages (DSLs). Applications use these DSLs to customize system policies and add new functionality. For performance, the kernel executes them via just-in-time (JIT) compilation. The correctness of these JITs is crucial for the security of the kernel: bugs in in-kernel JITs have led to numerous critical issues and patches.

This chapter presents JitSynth, the first tool for synthesizing verified JITs for in-kernel DSLs. JitSynth takes as input interpreters for the source DSL and the target instruction set architecture. Given these interpreters, and a mapping from source to target states, JitSynth synthesizes a verified JIT compiler from the source to the target. Our key idea is to formulate this synthesis problem as one of synthesizing a per-instruction compiler for *abstract register machines*. Our core technical contribution is a new *compiler metasketch* that enables JitSynth to efficiently explore the resulting synthesis search space. To evaluate JitSynth, we use it to synthesize a JIT from eBPF to RISC-V and compare to a recently developed Linux JIT. The synthesized JIT avoids all known

bugs in the Linux JIT, with an average slowdown of 1.82× in the performance of the generated code. We also use JITSYNTH to synthesize JITs for two additional source-target pairs. The results show that JITSYNTH offers a promising new way to develop verified JITs for in-kernel DSLs.

2.1 Overview

Modern operating systems (OSes) can be customized with user-specified programs that implement functionality like system call whitelisting, performance profiling, and power management [Eng96, Fle17, MJ93]. For portability and safety, these programs are written in restricted domain-specific languages (DSLs), and the kernel executes them via interpretation and, for better performance, just-in-time (JIT) compilation. The correctness of in-kernel interpreters and JITs is crucial for the reliability and security of the kernel, and bugs in their implementations have led to numerous critical issues and patches [Hor18, Pau20]. More broadly, embedded DSLs are also used to customize—and compromise [Bla10, KHF+19]—other low-level software, such as font rendering and anti-virus engines [CCK+13]. Providing formal guarantees of correctness for in-kernel DSLs is thus a pressing practical and research problem with applications to a wide range of systems software.

Prior work has tackled this problem through interactive theorem proving. For example, the Jitk framework [WLZ+14] uses the Coq interactive theorem prover [The20] to implement and verify the correctness of a JIT compiler for the classic Berkeley Packet Filter (BPF) language [MJ93] in the Linux kernel. But such an approach presents two key challenges. First, Jitk imposes a significant burden on DSL developers, requiring them to implement both the interpreter and the JIT compiler in Coq, and then manually prove the correctness of the JIT compiler with respect to the interpreter. Second, the resulting JIT implementation is extracted from Coq into OCaml and cannot be run in the kernel; rather, it must be run in user space, sacrificing performance and enlarging the trusted computing base (TCB) by relying on the OCaml runtime as part of the TCB.

This chapter addresses these challenges with JitSynth, the first tool for synthesizing verified JIT compilers for in-kernel DSLs. JitSynth takes as input interpreters for the source DSL and the target instruction set architecture (ISA), and it synthesizes a JIT compiler that is guaranteed to transform each source program into a semantically equivalent target program. Using JitSynth, DSL developers write no proofs or compilers. Instead, they write the semantics of the source and target languages in the form of interpreters and a mapping from source to target states, which JitSynth trusts to be correct. The synthesized JIT compiler is implemented in C; thus, it can run directly in the kernel.

At first glance, synthesizing a JIT compiler seems intractable. Even the simplest compiler contains thousands of instructions, whereas existing synthesis techniques scale to tens of instructions. To tackle this problem in our setting, we observe that in-kernel DSLs are similar to ISAs: both take the form of bytecode instructions for an *abstract register machine*, a simple virtual machine with a program counter, a few registers, and limited memory store [WLZ+14]. We also observe that in practice, the target machine has at least as many resources (registers and memory) as the source machine; and that JIT compilers for such abstract register machines perform register allocation statically at compile time. Our main insight is that we can exploit these properties to make synthesis tractable through *decomposition* and *prioritization*, while preserving soundness and completeness.

JitSynth works by decomposing the JIT synthesis problem into the problem of synthesizing individual *mini compilers* for every instruction in the source language. Each mini compiler is synthesized by generating a *compiler metasketch* [BTGC16], a set of ordered sketches that collectively represent *all* instruction sequences in the target ISA. These sketches are then solved by an off-the-shelf synthesis tool based on reduction to SMT [TB14]. The synthesis tool ensures that the target instruction sequence is semantically equivalent to the source instruction, according to the input interpreters. The order in which the sketches are explored is key to making this search practical, and JitSynth contributes two techniques for biasing the search towards tightly

constrained, and therefore tractable, sketches that are likely to contain a correct program.

First, we observe that source instructions can often be implemented with target instructions that access the same parts of the state (e.g., only registers). Based on this observation, we develop *read-write sketches*, which restrict the synthesis search space to a subset of the target instructions, based on a sound and precise summary of their semantics. Second, we observe that hand-written JITs rely on pseudoinstructions to generate common target sequences, such as loading immediate (constant) values into registers. We use this observation to develop *pre-load sketches*, which employ synthesized pseudoinstructions to eliminate the need to repeatedly search for common target instruction subsequences.

We have implemented JITSYNTH in Rosette [TB14] and used it to synthesize JIT compilers for three widely used in-kernel DSLs. As our main case study, we used JITSYNTH to synthesize a RISC-V [RIS19] compiler for extended BPF (eBPF) [Fle17], an extension of classic BPF [MJ93], used by the Linux kernel. Concurrently with our work, Linux developers manually built a JIT compiler for the same source and target pair, and a team of researchers found nine correctness bugs in that compiler shortly after its release [NBG+19]. In contrast, our JIT compiler is verified by construction; it supports 87 out of 102 eBPF instructions and passes all the Linux kernel tests within this subset, including the regression tests for these nine bugs. Our synthesized compiler generates code that is 5.24× faster than interpreted code and 1.82× times slower than the code generated by the Linux JIT. We also used JITSYNTH to synthesize a JIT from libseccomp [Edg12], a policy language for system call whitelisting, to eBPF, and a JIT from classic BPF to eBPF. The synthesized JITs avoid previously found bugs in the existing generators for these source target pairs, while incurring, on average, a 2.28–2.61× slowdown in the performance of the generated code.

To summarize, this chapter makes the following contributions:

1. JITSYNTH, the first tool for synthesizing verified JIT compilers for in-kernel DSLs, given

the semantics of the source and target languages as interpreters.

2. A novel formulation of the JIT synthesis problem as one of synthesizing a per-instruction compiler for *abstract register machines*.

3. A novel *compiler metasketch* that enables JitSynth to solve the JIT synthesis problem with an off-the-shelf synthesis engine.

4. An evaluation of JitSynth's effectiveness, showing that it can synthesize verified JIT compilers for three widely used in-kernel DSLs.

The rest of this chapter is organized as follows. Section 2.2 illustrates JitSynth on a small example. Section 2.3 formalizes the JIT synthesis problem for in-kernel DSLs. Section 2.4 presents the JitSynth algorithm for generating and solving compiler metasketches. Section 2.5 provides implementation details. Section 2.6 evaluates JitSynth. Section 2.7 discusses related work. Section 2.8 concludes.

2.2 JitSynth in a nutshell

This section provides an overview of JitSynth by illustrating how it synthesizes a toy JIT compiler (Figure 2.1). The source language of the JIT is a tiny subset of eBPF [Fle17] consisting of one instruction, and the target language is a subset of 64-bit RISC-V [RIS19] consisting of seven instructions. Despite the simplicity of our languages, the Linux kernel JIT used to produce incorrect code for this eBPF instruction [Nel19]; such miscompilation bugs not only lead to correctness issues, but also enable adversaries to compromise the OS kernel by crafting malicious eBPF programs [WLZ+14]. This section shows how JitSynth can be used to synthesize a JIT that is verified with respect to the semantics of the source and target languages.

instruction	description	semantics
eBPF (subset):		
`addi32` *dst*, *imm32*	32-bit add (high 32 bits cleared)	$R[dst] \leftarrow 0^{32} \oplus (\text{extract}(31, 0, R[dst]) + imm32)$
RISC-V (subset):		
`lui` *rd*, *imm20*	load upper immediate	$R[rd] \leftarrow \text{sext64}(imm20 \oplus 0^{12})$
`addiw` *rd*, *rs*, *imm12*	32-bit register-immediate add	$R[rd] \leftarrow \text{sext64}(\text{extract}(31, 0, R[rs]) + \text{sext32}(imm12))$
`add` *rd*, *rs1*, *rs2*	register-register add	$R[rd] \leftarrow R[rs1] + R[rs2]$
`slli` *rd*, *rs*, *imm6*	register-immediate left shift	$R[rd] \leftarrow rs \ll (0^{58} \oplus imm6)$
`srli` *rd*, *rs*, *imm6*	register-immediate logical right shift	$R[rd] \leftarrow rs \gg (0^{58} \oplus imm6)$
`lb` *rd*, *rs*, *imm12*	load byte from memory	$R[rd] \leftarrow \text{sext64}(M[R[rs] + \text{sext64}(imm12)])$
`sb` *rs1*, *rs2*, *imm12*	store byte to memory	$M[R[rs1] + \text{sext64}(imm12)] \leftarrow \text{extract}(7, 0, R[rs2])$

Figure 2.1: Subsets of eBPF and RISC-V used as source and target languages, respectively, in our running example: $R[r]$ denotes the value of register r; $M[a]$ denotes the value at memory address a; $\oplus$ denotes concatenation of bitvectors; superscripts (e.g., 0^{32}) denote repetition of bits; $\text{sext32}(x)$ and $\text{sext64}(x)$ sign-extend x to 32 and 64 bits, respectively; and $\text{extract}(i, j, x)$ produces a subrange of bits of x from index i down to j.

In-kernel languages. JITSYNTH expects the source and target languages to be a set of instructions for manipulating the state of an *abstract register machine* (Section 2.3). This state consists of a program counter (*pc*), a finite sequence of general-purpose registers (*reg*), and a finite sequence of memory locations (*mem*), all of which store bitvectors (i.e., finite precision integers). The length of these bitvectors is defined by the language; for example, both eBPF and RISC-V store 64-bit values in their registers. An instruction consists of an *opcode* and a finite set of *fields*, which are bitvectors representing either register identifiers or immediate (constant) values. For instance, the `addi32` instruction in eBPF has two fields: *dst* is a 4-bit value representing the index of the output register, and *imm32* is a 32-bit immediate. (eBPF instructions may have two additional fields *src* and *off*, which are not shown here as they are not used by `addi32`.) An abstract register machine for a language gives meaning to its instructions: the machine consumes an instruction and a state, and produces a state that is the result of executing that instruction. Figure 2.1 shows a high-level description of the abstract register machines for our languages.

JitSynth interface. To synthesize a compiler from one language to another, JitSynth takes as input their syntax, semantics, and a mapping from source to target states. All three inputs are given as a program in a *solver-aided host language* [TB14]. JitSynth uses Rosette as its host, but the host can be any language with a symbolic evaluation engine that can reduce the semantics of host programs to SMT constraints (e.g., [SLTB+06]). Figure 2.2 shows the interpreters for the source and target languages (i.e., emulators for their abstract register machines), as well as the state-mapping functions `regST`, `pcST`, and `memST` that JitSynth uses to determine whether a source state σ_S is equivalent to a target state σ_T. In particular, JitSynth deems these states equivalent, denoted by $\sigma_S \cong \sigma_T$, whenever $reg(\sigma_T)[\texttt{regST}(r)] = reg(\sigma_S)[r]$, $pc(\sigma_T) = \texttt{pcST}(pc(\sigma_S))$, and $mem(\sigma_T)[\texttt{memST}(a)] = mem(\sigma_S)[a]$ for all registers r and memory addresses a.

```
(struct state (regs mem pc) #:transparent)        ; Abstract register machine state.
                                                  ; Input 1/3: toy eBPF.
(struct ebpf-insn (opcode dst src off imm))       ;  - eBPF instruction format;
(define (ebpf-interpret insn st)                  ;  - eBPF interpreter for addi32.
  (define-match (ebpf-insn op dst _ _ imm) insn)  ;
  (case op                                        ; Note: addi32 does not use the src
    [(addi32)                                     ; and off fields.
     (state
      (reg-set st dst (concat (bv 0 32) (bvadd (extract 31 0 (reg-ref st dst)) imm)))
      (state-mem st)
      (bvadd (state-pc st) (bv 1 64)))]))
                                                  ; Input 2/3: toy RISC-V.
(struct rv-insn (opcode rd rs1 rs2 imm))          ;  - RISC-V instruction format;
(define (rv-interpret insn st)                    ;  - RISC-V interpreter.
  (define-match (rv-insn op rd rs1 rs2 imm) insn)
  (case op
    [(lui)
     (state
      (reg-set st rd (sext64 (concat imm (bv 0 12))))
      (state-mem st)
      (bvadd (state-pc st) (bv 4 64)))] ...))
                                                  ; Input 3/3: state mapping functions.
(define (regST r)                                 ;  - Register mapping:
  (cond [(equal? r (bv 0 4)) (bv 15 5)] ...))     ;    - eBPF r0 -> RISC-V x15, ...;
(define (memST a) a)                              ;  - Memory mapping is the identity.
(define (pcST pc) (bvshl pc (bv 2 64)))           ;  - PC mapping.
```

Figure 2.2: Snippets of inputs to JitSynth: the interpreters for the source (eBPF) and and target (RISC-V) languages and state-mapping functions.

Decomposition into per-instruction compilers. Given these inputs, JITSYNTH generates a *per-instruction compiler* from the source to the target language. To ensure that the resulting compiler is correct (Theorem 2.1), and that one will be found if it exists (Theorem 2.2), JITSYNTH puts two restrictions on its inputs. First, the inputs must be self-finitizing [TB14], meaning that both the interpreters and the mapping functions must have a finite symbolic execution tree when applied to symbolic inputs. Second, the target machine must have at least as many registers and memory locations as the source machine; these storage cells must be as wide as those of the source machine; and the state-mapping functions (`pcST`, `regST`, and `memST`) must be injective. Our toy inputs satisfy these restrictions, as do the real in-kernel languages evaluated in Section 2.6.

Synthesis workflow. JITSYNTH generates a per-instruction compiler for a given source and target pair in two stages. The first stage uses an optimized *compiler metasketch* to synthesize a mini compiler from every instruction in the source language to a sequence of instructions in the target language (Section 2.4). The second stage then simply stitches these mini compilers into a full C compiler using a trusted outer loop and a switch statement. The first stage is a core technical contribution of this chapter, and we illustrate it next on our toy example.

Metasketches. To understand how JITSYNTH works, consider the basic problem of determining if every `addi32` instruction can be emulated by a sequence of k instructions in toy RISC-V. In particular, we are interested in finding a program $C_{\texttt{addi32}}$ in our host language (which JITSYNTH translates to C) that takes as input a source instruction s = `addi32` *dst*, *imm32* and outputs a semantically equivalent RISC-V program $t = [t_1, \ldots, t_k]$. That is, for all *dst*, *imm32*, and for all equivalent states $\sigma_S \cong \sigma_T$, we have $run(s, \sigma_S, \texttt{ebpf-interpret}) \cong run(t, \sigma_T, \texttt{rv-interpret})$, where $run(e, \sigma, f)$ executes the instruction interpreter f on the sequence of instructions e, starting from the state σ (Definition 2.3).

We can solve this problem by asking the host synthesizer to search for $C_{\texttt{addi32}}$ in a space of candidate mini compilers of length k. We describe this space with a syntactic template, or a *sketch*, as shown in Figure 2.3.

In this sketch, `(??insn dst imm)` stands for a missing expression—a hole—that the synthesizer needs to fill with an instruction from the toy RISC-V language. To fill an instruction hole, the synthesizer must find an expression that computes the value of the target instruction's fields. JitSynth limits this expression language to bitvector expressions (of any depth) over the fields of the source instruction and arbitrary bitvector constants.

Given this sketch, and our correctness specification for $C_{\texttt{addi32}}$, the synthesizer will search the space defined by the sketch for a program that satisfies the specification. Below is an example of the resulting toy compiler from eBPF to RISC-V, synthesized and translated to C by JitSynth (without the outer loop):

```
void compile(struct bpf_insn *insn, struct rv_insn *tgt_prog) {
  switch (insn->op) {
  case BPF_ADDI32:
    tgt_prog[0] = /* lui   x6, extract(19, 0, (imm + 0x800) >> 12) */
      rv_lui(6, extract(19, 0, (insn->imm + 0x800) >> 12));
    tgt_prog[1] = /* addiw x6, x6, extract(11, 0, imm) */
      rv_addiw(6, 6, extract(11, 0, insn->imm));
    tgt_prog[2] = /* add   rd, rd, x6 */
      rv_add(regmap(insn->dst), regmap(insn->dst), 6);
    tgt_prog[3] = /* slli  rd, rd, 32 */
      rv_slli(regmap(insn->dst), regmap(insn->dst), 32);
    tgt_prog[4] = /* srli  rd, rd, 32 */
      rv_srli(regmap(insn->dst), regmap(insn->dst), 32);
    break;
  }
}
```

Once we know how to synthesize a compiler of length k, we can easily extend this solution into a naive method for synthesizing a compiler of any length. We simply enumerate sketches of increasing lengths, $k = 1, 2, 3, \ldots$, invoke the synthesizer on each generated sketch, and stop as soon as a solution is found (if ever). The resulting ordered set of sketches forms a metasketch [BTGC16]—i.e., a search space and a strategy for exploring it—that contains all candidate mini compilers (in a

```
(define (compile-addi32 s)       ; Returns a list of k instruction holes, to be
  (define dst (ebpf-insn-dst s)) ; filled with toy RISC-V instructions. Each
  (define imm (ebpf-insn-imm s)) ; hole represents a set of choices, defined
  (list (??insn dst imm) ...))   ; by the ??insn procedure.

(define (??insn . sf)            ; Takes as input source instruction fields and
  (define rd  (??reg sf))        ; uses them to construct target field holes.
  (define rs1 (??reg sf))        ; ??reg and ??imm field holes are bitvector
  (define rs2 (??reg sf))        ; expressions over sf and arbitrary constants.
  (choose*                       ; Returns an expression that chooses among
    (rv-insn lui rd rs1 rs2 (??imm 20 sf))   ; lui, addiw,
    ...                                      ; ..., and
    (rv-insn sb  rd rs1 rs2 (??imm 12 sf)))) ; sb instructions.
```

Figure 2.3: A fixed-length sketch describing the space of candidate mini compilers for the eBPF `addi32` instruction. (`??insn dst imm`) describes a missing expression that the synthesizer must fill with a RISC-V instruction.

subset of the host language) from the source to the target language. This naive metasketch can be used to find a mini compiler for our toy example in 493 minutes. However, it fails to scale to real in-kernel DSLs (Section 2.6), motivating the need for JITSYNTH's optimized compiler metasketches.

Compiler metasketches. JITSYNTH optimizes the naive metasketch by extending it with two kinds of more tightly constrained sketches, which are explored first. A constrained sketch of size k usually contains a correct solution of a given size if one exists, but if not, JITSYNTH will eventually explore the naive sketch of the same length, to maintain completeness. We give the intuition behind the two optimizations here, and present them in detail in Section 2.4.

First, we observe that practical source and target languages include similar kinds of instructions. For example, both eBPF and RISC-V include instructions for adding immediate values to registers. This similarity often makes it possible to emulate a source instruction with a sequence of target instructions that access the same part of the state (the program counter, registers, or memory) as the source instruction. For example, `addi32` reads and writes only registers, not memory, and it can be emulated with RISC-V instructions that also access only registers. To exploit

this observation, we introduce *read-write sets*, which summarize, soundly and precisely, how an instruction accesses state. JITSYNTH uses these sets to define *read-write sketches* for a given source instruction, including only target instructions that access the state in the same way as the source instruction. For instance, a read-write sketch for `addi32` excludes both `lb` and `sb` instructions because they read and write memory as well as registers.

Second, we observe that hand-written JITs use pseudoinstructions to simplify their implementation of mini compilers. These are simply subroutines or macros for generating target sequences that implement common functionality. For example, the Linux JIT from eBPF to RISC-V includes a pseudoinstruction for loading 32-bit immediates into registers. JITSYNTH mimics the way hand-written JITs use pseudoinstructions with the help of *pre-load sketches*. These sketches first use a synthesized pseudoinstruction to create a sequence of concrete target instructions that load source immediates into scratch registers; then, they include a compute sequence comprised of read-write instruction holes. Applying these optimizations to our toy example, JITSYNTH finds a mini compiler for `addi32` in 5 seconds—a roughly 6000× speedup over the naive metasketch.

2.3 Problem Statement

This section formalizes the compiler synthesis problem for in-kernel DSLs. We focus on JIT compilers, which, for our purposes, means one-pass compilers [Eng96]. To start, we define *abstract register machines* as a way to specify the syntax and semantics of in-kernel languages. Next, we formulate our compiler synthesis problem as one of synthesizing a set of sound *mini compilers* from a single source instruction to a sequence of target instructions. Finally, we show that these mini compilers compose into a sound JIT compiler, which translates every source program into a semantically equivalent target program.

Abstract register machines. An abstract register machine (ARM) provides a simple interface for specifying the syntax and semantics of an in-kernel language. The syntax is given as a set of abstract instructions, and the semantics is given as a transition function over instructions and machine states.

An *abstract instruction* (Definition 2.1) defines the name (*op*) and type signature ($\mathcal{F}$) of an operation in the underlying language. For example, the abstract instruction $(addi32, r \mapsto Reg, imm32 \mapsto BV(32))$ specifies the name and signature of the `addi32` operation from the eBPF language (Figure 2.1). Each abstract instruction represents the (finite) set of all *concrete instructions* that instantiate the abstract instruction's parameters with values of the right type. For example, `addi32` 0, 5 is a concrete instantiation of the abstract instruction for `addi32`. In the rest of this chapter, we will write "instruction" to mean a concrete instruction.

Definition 2.1 (Abstract and Concrete Instructions) *An* abstract instruction ι *is a pair* $(op, \mathcal{F})$ *where op is an opcode and* $\mathcal{F}$ *is a mapping from* fields *to their* types. *Field types include Reg, denoting register names, and* $BV(k)$*, denoting k-bit bitvector values. The abstract instruction* ι *represents all* concrete instructions $p = (op, F)$ *with the opcode op that bind each field* $f \in dom(\mathcal{F})$ *to a value* $F(f)$ *of type* $\mathcal{F}(f)$*. We write* $P(\iota)$ *to denote the set of all concrete instructions for* ι*, and we extend this notation to sets of abstract instructions in the usual way, i.e.,* $P(\mathcal{I}) = \bigcup_{\iota \in \mathcal{I}} P(\iota)$ *for the set* $\mathcal{I}$*.*

Instructions operate on machine *states* (Definition 2.2), and their semantics are given by the machine's *transition function* (Definition 2.3). A machine state consists of a program counter, a map from register names to register values, and a map from memory addresses to memory values. Each state component is either a bitvector or a map over bitvectors, making the set of all states of an ARM finite. The transition function of an ARM defines an interpreter for the ARM's language by specifying how to compute the output state for a given instruction and input state. We can apply this interpreter, together with the ARM's *fuel function*, to define an *execution* of the machine

on a program and an initial state. The fuel function takes as input a sequence of instructions and returns a natural number that bounds the number of steps (i.e., state transitions) the machine can make to execute the given sequence. The inclusion of fuel models the requirement of in-kernel languages for all program executions to terminate [WLZ+14]. It also enables us to use symbolic execution to soundly reduce the semantics of these languages to SMT constraints, in order to formulate the synthesis queries in Section 2.4.5.

Definition 2.2 (State) *A* state σ *is a tuple* (pc, reg, mem) *where pc is a value, reg is a function from register names to values, and mem is a function from memory addresses to values. Register names, memory addresses, and all values are finite-precision integers, or bitvectors. We write* $|\sigma|$ *to denote the* size *of the state* σ*. The size* $|\sigma|$ *is defined to be the tuple* $(r, m, k_{pc}, k_{reg}, k_{mem})$*, where* r *is the number of registers in* σ*,* m *is the number of memory addresses, and* k_{pc}*,* k_{reg}*, and* k_{mem} *are the width of the bitvector values stored in the pc, reg, and mem, respectively. Two states have the same size if* $|\sigma_i| = |\sigma_j|$*; one state is smaller than another,* $|\sigma_i| \leq |\sigma_j|$*, if each element of* $|\sigma_i|$ *is less than or equal to the corresponding element of* $|\sigma_j|$*.*

Definition 2.3 (Abstract Register Machines and Executions) *An* abstract register machine $\mathcal{A}$ *is a tuple* $(\mathcal{I}, \Sigma, \mathcal{T}, \Phi)$ *where* $\mathcal{I}$ *is a set of abstract instructions,* Σ *is a set of states of the same size,* $\mathcal{T} : P(\mathcal{I}) \to \Sigma \to \Sigma$ *is a* transition function *from instructions and states to states, and* $\Phi : List(P(\mathcal{I})) \to \mathbb{N}$ *is a* fuel function *from sequences of instructions to natural numbers. Given a state* $\sigma_0 \in \Sigma$ *and a sequence of instructions* $\boldsymbol{p}$ *drawn from* $P(\mathcal{I})$*, we define the* execution *of* $\mathcal{A}$ *on* $\boldsymbol{p}$ *and* σ_0 *to be the result of applying* $\mathcal{T}$ *to* $\boldsymbol{p}$ *at most* $\Phi(\boldsymbol{p})$ *times. That is,* $\mathcal{A}(\boldsymbol{p}, \sigma_0) = run(\boldsymbol{p}, \sigma_0, \mathcal{T}, \Phi(\boldsymbol{p}))$*, where*

$$run(\boldsymbol{p}, \sigma, \mathcal{T}, k) = \begin{cases} \sigma, & \text{if } k = 0 \text{ or } pc(\sigma) \notin [0, |\boldsymbol{p}|) \\ run(\boldsymbol{p}, \mathcal{T}(\boldsymbol{p}[pc(\sigma)], \sigma), \mathcal{T}, k - 1), & \text{otherwise.} \end{cases}$$

Synthesizing JIT compilers for ARMs. Given a source and target ARM, our goal is to synthesize a one-pass JIT compiler that translates source programs to semantically equivalent target programs. To make synthesis tractable, we fix the structure of the JIT to consist of an outer loop and a switch statement that dispatches compilation tasks to a set of *mini compilers* (Definition 2.4). Our synthesis problem is therefore to find a sound mini compiler for each abstract instruction in the source machine (Definition 2.5).

Definition 2.4 (Mini Compiler) *Let* $\mathcal{A}_S = (\mathcal{I}_S, \Sigma_S, \mathcal{T}_S, \Phi_S)$ *and* $\mathcal{A}_T = (\mathcal{I}_T, \Sigma_T, \mathcal{T}_T, \Phi_T)$ *be two abstract register machines,* $\cong$ *an equivalence relation on their states* Σ_S *and* Σ_T*, and* $C : P(\iota) \rightarrow List(P(\mathcal{I}_T))$ *a function for some* $\iota \in \mathcal{I}_S$*. We say that* C *is a* sound mini compiler *for* ι *with respect to* $\cong$ *iff*

$$\forall \sigma_S \in \Sigma_S,\ \sigma_T \in \Sigma_T,\ p \in P(\iota).\ \sigma_S \cong \sigma_T \Rightarrow \mathcal{A}_S(p, \sigma_S) \cong \mathcal{A}_T(C(p), \sigma_T)$$

Definition 2.5 (Mini Compiler Synthesis) *Given two abstract register machines* $\mathcal{A}_S = (\mathcal{I}_S, \Sigma_S, \mathcal{T}_S, \Phi_S)$ *and* $\mathcal{A}_T = (\mathcal{I}_T, \Sigma_T, \mathcal{T}_T, \Phi_T)$*, as well as an equivalence relation* $\cong$ *on their states, the* mini compiler synthesis problem *is to generate a sound mini compiler* C_ι *for each* $\iota \in \mathcal{I}_S$ *with respect to* $\cong$*.*

The general version of our synthesis problem, defined above, uses an arbitrary equivalence relation $\cong$ between the states of the source and target machines to determine if a source and target program are semantically equivalent. JITSYNTH can, in principle, solve this problem with the naive metasketch described in Section 2.2. In practice, however, the naive metasketch scales poorly, even on small languages such as toy eBPF and RISC-V. So, in this work, we focus on source and target ARMs that satisfy an additional assumption on their state equivalence relation: it can be expressed in terms of injective mappings from source to target states (Definition 2.6). This restriction enables JITSYNTH to employ optimizations (such as pre-load sketches described

in Section 2.4.4) that are crucial to scaling synthesis to real in-kernel languages.

Definition 2.6 (Injective State Equivalence Relation) *Let $\mathcal{A}_S$ and $\mathcal{A}_T$ be abstract register machines with states Σ_S and Σ_T such that $|\sigma_S| \leq |\sigma_T|$ for all $\sigma_S \in \Sigma_S$ and $\sigma_T \in \Sigma_T$. Let $\mathcal{M}$ be a* state mapping $(\mathcal{M}_{pc}, \mathcal{M}_{reg}, \mathcal{M}_{mem})$ *from Σ_S and Σ_T, where $\mathcal{M}_{pc}$ multiplies the program counter of the states in Σ_S by a constant factor, $\mathcal{M}_{reg}$ is an injective map from register names in Σ_S to those in Σ_T, and $\mathcal{M}_{mem}$ is an injective map from memory addresses in Σ_S to those in Σ_T. We say that two states $\sigma_S \in \Sigma_S$ and $\sigma_T \in \Sigma_T$ are equivalent according to $\mathcal{M}$, written $\sigma_S \cong_{\mathcal{M}} \sigma_T$, iff $\mathcal{M}_{pc}(pc(\sigma_S)) = pc(\sigma_T)$, $reg(\sigma_S)[r] = reg(\sigma_T)[\mathcal{M}_{reg}(r)]$ for all register names $r \in \mathrm{dom}(reg(\sigma_S))$, and $mem(\sigma_S)[a] = mem(\sigma_T)[\mathcal{M}_{mem}(a)]$ for all memory addresses $a \in \mathrm{dom}(mem(\sigma_S))$. The binary relation $\cong_{\mathcal{M}}$ is called an* injective state equivalence relation *on $\mathcal{A}_S$ and $\mathcal{A}_T$.*

Soundness of JIT compilers for ARMs. Finally, we note that a JIT compiler composed from the synthesized mini compilers correctly translates every source program to an equivalent target program. We formulate and prove this theorem using the Lean theorem prover [dMKA+15].

Theorem 2.1 (Soundness of JIT compilers) *Let $\mathcal{A}_S = (\mathcal{I}_S, \Sigma_S, \mathcal{T}_S, \Phi_S)$ and $\mathcal{A}_T = (\mathcal{I}_T, \Sigma_T, \mathcal{T}_T, \Phi_T)$ be abstract register machines, $\cong_{\mathcal{M}}$ an injective state equivalence relation on their states such that $M_{pc}(pc(\sigma_S)) = N_{pc} pc(\sigma_S)$, and $\{C_1, \ldots, C_{|\mathcal{I}_S|}\}$ a solution to the mini compiler synthesis problem for $\mathcal{A}_S$, $\mathcal{A}_T$, and $\cong_{\mathcal{M}}$ where $\forall s \in P(\iota).\ |C_i(s)| = N_{pc}$. Let $C : P(\mathcal{I}_S) \to List(P(\mathcal{I}_T))$ be a function that maps concrete instructions $s \in P(\iota)$ to the compiler output $C_\iota(s)$ for $\iota \in \mathcal{I}_S$. If $\boldsymbol{s} = s_1, \ldots, s_n$ is a sequence of concrete instructions drawn from $\mathcal{I}_S$, and $\boldsymbol{t} = C(s_1) \cdot \ldots \cdot C(s_n)$ where $\cdot$ stands for sequence concatenation, then $\forall \sigma_S \in \Sigma_S, \sigma_T \in \Sigma_T.\ \sigma_S \cong_{\mathcal{M}} \sigma_T \Rightarrow \mathcal{A}_S(\boldsymbol{s}, \sigma_S) \cong_{\mathcal{M}} \mathcal{A}_T(\boldsymbol{t}, \sigma_T)$.*

2.4 Solving the Mini Compiler Synthesis Problem

This section presents our approach to solving the mini compiler synthesis problem defined in Section 2.3. We employ syntax-guided synthesis [SLTB+06] to search for an implementation of a mini compiler in a space of candidate programs. Our core contribution is an effective way to structure this space using a *compiler metasketch*. This section presents our algorithm for generating compiler metasketches, describes its key subroutines and optimizations, and shows how to solve the resulting sketches with an off-the-shelf synthesis engine.

2.4.1 Generating Compiler Metasketches

JITSYNTH synthesizes mini compilers by generating and solving *metasketches* [BTGC16]. A metasketch describes a space of candidate programs using an ordered set of syntactic templates or *sketches* [SLTB+06]. These sketches take the form of programs with missing expressions or *holes*, where each hole describes a finite set of candidate completions. JITSYNTH sketches are expressed in a *host language* $\mathcal{H}$ that serves both as the implementation language for mini compilers and the specification language for ARMs. JITSYNTH expects the host to provide a synthesizer for completing sketches and a symbolic evaluator for reducing ARM semantics to SMT constraints. JITSYNTH uses these tools to generate optimized metasketches for mini compilers, which we call *compiler metasketches*.

Figure 2.4 shows our algorithm for generating compiler metasketches. The algorithm, CMS, takes as input an abstract source instruction ι for a source machine $\mathcal{A}_S$, a target machine $\mathcal{A}_T$, and a state mapping $\mathcal{M}$ from $\mathcal{A}_S$ to $\mathcal{A}_T$. Given these inputs, it lazily enumerates an infinite set of *compiler sketches* that collectively represent the space of all straight-line bitvector programs from $P(\iota)$ to $List(P(\mathcal{I}_T))$. In particular, each compiler sketch consists of k target *instruction holes*, constructed from field holes that denote bitvector expressions (over the fields of ι) of depth d or

```
1  function CMS(ι, 𝒜_S, 𝒜_T, ℳ)                          ▷ ι ∈ I_S, 𝒜_S = (I_S, ...)
2      for n ∈ ℤ⁺ do                               ▷ Lazily enumerates all compiler sketches
3          for k ∈ [1, n], d = n − k do                      ▷ of length k and depth d,
4              yield PLD(k, d, ι, 𝒜_S, 𝒜_T, ℳ)            ▷ yielding the pre-load sketch first,
5              yield RW(k, d, ι, 𝒜_S, 𝒜_T, ℳ)               ▷ read-write sketch next, and
6              yield Naive(k, d, ι, 𝒜_S, 𝒜_T, ℳ)          ▷ the most general sketch last.
7          end for
8      end for
9  end function
```

Figure 2.4: Compiler metasketch for the abstract source instruction ι, source machine $\mathcal{A}_S$, target machine $\mathcal{A}_T$, and state mapping $\mathcal{M}$ from $\mathcal{A}_S$ to $\mathcal{A}_T$.

less. For each length k and depth d, the CMS loop generates three kinds of compiler sketches: the *pre-load*, the *read-write*, and the *naive* sketch. The naive sketch (Section 2.4.2) is the most general, consisting of all candidate mini compilers of length k and depth d. But it also scales poorly, so CMS first yields the pre-load (Section 2.4.4) and read-write (Section 2.4.3) sketches. As we will see later, these sketches describe a subset of the programs in the naive sketch, and they are designed to prioritize exploring small parts of the search space that are likely to contain a correct mini compiler for ι, if one exists.

2.4.2 Generating Naive Sketches

The most general sketch we consider, $\textsc{Naive}(k, d, \iota, \mathcal{A}_S, \mathcal{A}_T, \mathcal{M})$, is shown in Figure 2.5. This sketch consists of k instruction holes that can be filled with any instruction from $\mathcal{I}_T$. An instruction hole chooses between expressions of the form (op_T, H), where op_T is a target opcode, and H specifies the field holes for that opcode. Each field hole is a bitvector expression (of depth d) over the fields of the input source instruction and arbitrary bitvector constants. This lets target instructions use the immediates and registers (modulo $\mathcal{M}$) of the source instruction, as well as arbitrary constant values and register names. Letting field holes include constant register names allows the synthesized mini compilers to use target registers unmapped by $\mathcal{M}$ as temporary, or

```
1  function NAIVE(k, d, ι, 𝒜_S, 𝒜_T, ℳ)                          ▷ ι ∈ ℐ_S, 𝒜_S = (ℐ_S, ...)
2      (op, ℱ) ← ι, (ℐ_T, ...) ← 𝒜_T                             ▷ Source instruction, target instructions.
3      p ← FreshId()                                             ▷ Identifier for the compiler's input.
4      body ← []                                                 ▷ The body of the compiler is a sequence
5      for 0 ≤ i < k do                                          ▷ of k target instruction holes.
6          I ← {}                                                ▷ The set I of choices for a target instruction hole
7          for (op_T, ℱ_T) ∈ ℐ_T do                              ▷ includes all instructions from ℐ_T.
8              E ← {Expr(p.f, ℳ) | f ∈ dom(ℱ)}                   ▷ Any source field can appear in
9              H ← {f ↦ Field(ℱ_T(f), d, E) | f ∈ dom(ℱ_T)}      ▷ a target field hole, and
10             I ← I ∪ {Expr((op_T, H), ℳ)}                      ▷ any constant register or value.
11         end for
12         body ← body · [Choose(I)]                             ▷ Append a hole over I to the body.
13     end for
14     return Expr((λp ∈ P(ι) . body), ℳ)                        ▷ A mini compiler sketch for ι.
15 end function
```

Figure 2.5: Naive sketch of length k and maximum depth d for ι, $\mathcal{A}_S$, $\mathcal{A}_T$, and $\mathcal{M}$. Here, *Expr* creates an expression in the host language, using $\mathcal{M}$ to map from source to target register names and memory addresses; *Choose*(E) is a hole that chooses an expression from the set E; and *Field*(τ, d, E) is a hole for a bitvector expression of type τ and maximum depth d, constructed from arbitrary bitvector constants and expressions E.

scratch, storage. In essence, the naive sketch describes all straight-line compiler programs that can make free use of standard C arithmetic and bitwise operators, as well as scratch registers.

The space of such programs is intractably large, however, even for small inputs. For instance, it includes at least 2^{350} programs of length $k = 5$ and depth $d \leq 3$ for the toy example from Section 2.2. JITSYNTH therefore employs two effective heuristics to direct the exploration of this space toward the most promising candidates first, as defined by the read-write and pre-load sketches.

2.4.3 Generating Read-Write Sketches

The read-write sketch, $\text{RW}(k, d, \iota, \mathcal{A}_S, \mathcal{A}_T, \mathcal{M})$, is based on the observation that many practical source and target languages provide similar functionality, so a source instruction ι can often be emulated with target instructions that access the same parts of the state as ι. For example, the

ι	*Read*(ι)	*Write*(ι)	*Write*(ι, field)
`addi32`	$\{L_{reg}\}$	$\{L_{reg}\}$	*imm*: $\{L_{reg}\}$; *off*: $\emptyset$; *src*: $\emptyset$; *dst*: $\{L_{reg}\}$
`lui`	$\{L_{reg}\}$	$\{L_{reg}\}$	*rd*: $\{L_{reg}\}$; *imm20*: $\{L_{reg}\}$
`sb`	$\{L_{reg}\}$	$\{L_{mem}\}$	*rs1*: $\{L_{mem}\}$; *rs2*: $\{L_{mem}\}$; *imm12*: $\{L_{mem}\}$

Figure 2.6: Read and write sets for the `addi32`, `lui`, and `sb` instructions from Figure 2.1.

`addi32` instruction from eBPF reads and writes only registers (not, e.g., memory), and it can be emulated with RISC-V instructions that also touch only registers (Section 2.2). Moreover, note that the semantics of `addi32` ignores the values of its *src* and *off* fields, and that the target RISC-V instructions do the same. Based on these observations, our optimized sketch for `addi32` would therefore consists of instruction holes that allow only register-register instructions, with field holes that exclude *src* and *off*. We first formalize this intuition with the notion of *read and write sets*, and then describe how JitSynth applies such sets to create RW sketches.

Read and write sets. Read and write sets provide a compact way to summarize the semantics of an abstract instruction ι. This summary consists of a set of *state labels*, where a state label is one of L_{reg}, L_{mem}, and L_{pc} (Definition 2.7). Each label in a summary set represents a state component (registers, memory, or the program counter) that a concrete instance of ι may read or write during some execution. We compute three such sets of labels for every ι: the read set *Read*(ι), the write set *Write*(ι), and the write set *Write*(ι, f) for each field f of ι. Figure 2.6 shows these sets for the toy eBPF and RISC-V instructions.

The read set *Read*(ι) specifies which components of the input state may affect the execution of ι (Definition 2.8). For example, if *Read*(ι) includes L_{reg}, then some concrete instance of ι produces different output states when executed on two input states that differ only in register values. The write set *Write*(ι) specifies which components of the output state may be affected by executing ι (Definition 2.9). In particular, if *Write*(ι) includes L_{reg} (or L_{mem}), then executing some concrete

instance of ι on an input state produces an output state with different register (or memory) values. The inclusion of L_{pc} is based on a separate condition, designed to distinguish jump instructions from fall-through instructions. Both kinds of instructions change the program counter, but fall-through instructions always change it in the same way. So, $L_{pc} \in \mathit{Write}(\iota)$ if two instances of ι can write different values to the program counter. Finally, the field write set, $\mathit{Write}(\iota, f)$, specifies the parts of the output state are affected by the value of the field f; $L_n \in \mathit{Write}(\iota, f)$ means that two instances of ι that differ only in f can produce different outputs when applied to the same input state.

JITSYNTH computes all read and write sets from their definitions, by using the host symbolic evaluator to reduce the reasoning about instruction semantics to SMT queries. This reduction is possible because we assume that all ARM interpreters are self-finitizing, as discussed in Section 2.2.

Definition 2.7 (State Labels) *A* state label *is an identifier L_n where n is a state component, i.e., $n \in \{reg, mem, pc\}$. We write N for the set of all state components, and $\mathcal{L}$ for the set of all state labels. We also use state labels to access the corresponding state components: $L_n(\sigma) = n(\sigma)$ for all $n \in N$.*

Definition 2.8 (Read Set) *Let $\iota \in \mathcal{I}$ be an abstract instruction in $(\mathcal{I}, \Sigma, \mathcal{T}, \Phi)$. The* read set *of ι, $\mathit{Read}(\iota)$, is the set of all state labels $L_n \in \mathcal{L}$ such that $\exists p \in P(\iota).\, \exists L_w \in \mathit{Write}(\iota).\, \exists \sigma_a, \sigma_b \in \Sigma.\, (L_n(\sigma_a) \neq L_n(\sigma_b) \wedge (\bigwedge_{m \in N \setminus \{n\}} L_m(\sigma_a) = L_m(\sigma_b)) \wedge L_w(\mathcal{T}(p, \sigma_a)) \neq L_w(\mathcal{T}(p, \sigma_b)))$.*

Definition 2.9 (Write Set) *Let $\iota \in \mathcal{I}$ be an abstract instruction in $(\mathcal{I}, \Sigma, \mathcal{T}, \Phi)$. The* write set *of ι, $\mathit{Write}(\iota)$, includes the state label $L_n \in \{L_{reg}, L_{mem}\}$ iff $\exists p \in P(\iota).\, \exists \sigma \in \Sigma.\, L_n(\sigma) \neq L_n(\mathcal{T}(p, \sigma))$, and it includes the state label L_{pc} iff $\exists p_a, p_b \in P(\iota).\, \exists \sigma \in \Sigma.\, L_{pc}(\mathcal{T}(p_a, \sigma)) \neq L_{pc}(\mathcal{T}(p_b, \sigma))$.*

Definition 2.10 (Field Write Set) *Let f be a field of an abstract instruction $\iota = (op, \mathcal{F})$ in $(\mathcal{I}, \Sigma, \mathcal{T}, \Phi)$. The* write set *of ι and f, $\mathit{Write}(\iota, f)$, includes the state label $L_n \in \mathcal{L}$ iff $\exists p_a, p_b \in P(\iota).\, \exists \sigma \in \Sigma.\, (p_a.f \neq p_b.f) \wedge (\bigwedge_{g \in dom(\mathcal{F}) \setminus \{f\}} p_a.g = p_b.g) \wedge L_n(\mathcal{T}(p_a, \sigma)) \neq L_n(\mathcal{T}(p_b, \sigma))$, where $p.f$ denotes $F(f)$ for $p = (op, F)$.*

Using read and write sets. Given the read and write sets for a source instruction ι and target instructions $\mathcal{I}_T$, JitSynth generates the RW sketch of length k and depth d by modifying the Naive algorithm (Figure 2.5) as follows. First, it restricts each target instruction hole (line 7) to choose an instruction $\iota_T \in \mathcal{I}_T$ with the same read and write sets as ι, i.e., $Read(\iota) = Read(\iota_T)$ and $Write(\iota) = Write(\iota_T)$. Second, it restricts the target field holes (line 9) to use the source fields with the matching field write set, i.e., the hole for a target field f_T uses the source field f when $Write(\iota_T, f_t) = Write(\iota, f)$. For example, given the sets from Figure 2.6, the RW instruction holes for `addi32` exclude `sb` but include `lui`, and the field holes for `lui` use only the *dst* and *imm* source fields. More generally, the RW sketch for `addi32` consists of register-register instructions over *dst* and *imm*, as intended. This sketch includes 2^{290} programs of length $k = 5$ and depth $d \leq 3$, resulting in a 2^{60} fold reduction in the size of the search space compared to the Naive sketch of the same length and depth.

2.4.4 Generating Pre-Load Sketches

The pre-load sketch, PLD$(k, d, \iota, \mathcal{A}_S, \mathcal{A}_T, \mathcal{M})$, is based on the observation that hand-written JITs use macros or subroutines to generate frequently used target instruction sequences. For example, compiling a source instruction with immediate fields often involves loading the immediates into scratch registers, and hand-written JITs include a subroutine that generates the target instructions for performing these loads. The pre-load sketch shown in Figure 2.7 mimics this structure.

In particular, PLD generates a sequence of m concrete instructions that load the (used) immediate fields of ι, followed by a sequence of $k - m$ instruction holes. The instruction holes can refer to both the source registers (if any) and the scratch registers (via the arbitrary bitvector constants included in the *Field* holes). The function $Load(Expr(p.f), \mathcal{A}_T, \mathcal{M})$ returns a sequence of target instructions that load the immediate $p.f$ into an unused scratch register. This function itself is

synthesized by JITSYNTH using a variant of the RW sketch.

As an example, the pre-load sketch for `addi32` consists of two *Load* instructions (`lui` and `ad-diw` in the generated C code) and $k-2$ instruction holes. The holes choose among register-register instructions in toy RISC-V, and they can refer to the *dst* register of `addi32`, as well as any scratch register. The resulting sketch includes 2^{100} programs of length $k = 5$ and depth $d \leq 3$, providing a 2^{190} fold reduction in the size of the search space compared to the RW sketch.

```
1  function PLD(k, d, ι, 𝒜_S, 𝒜_T, ℳ)                          ▷ ι ∈ ℐ_S, 𝒜_S = (ℐ_S, ...)
2      (op, ℱ) ← ι, (I_T, ...) ← 𝒜_T                      ▷ Source instruction, target instructions.
3      p ← FreshId()                              ▷ Identifier for the compiler's input source instruction.
4      body ← []                                  ▷ The body of the compiler is a sequence with 2 parts:
5      imm ← {f | ℱ(f) = BV(k) and Write(ι, f) ≠ ∅}                    ▷ (1) Load each relevant
6      for f ∈ imm do                                   ▷ source immediate into a free scratch register
7          body ← body · Load(Expr(p.f), 𝒜_T, ℳ)              ▷ using the load pseudoinstruction.
8      end for
9      m ← |body|                                         ▷ Let m be the length of the load sequence.
10     if m ≥ k or m = 0 then return ⊥                      ▷ Return the empty sketch if m ∉ (0..k).
11     end if
12     for m ≤ i < k do                          ▷ (2) Create k − m target instruction holes, where the set
13         I ← {}                                       ▷ I of choices for a target instruction hole includes
14         for ι_T ∈ ℐ_T, ι_T = (op_T, ℱ_T) do                  ▷ all instructions from ℐ_T that read-write
15             rw_T ← Read(ι_T) × Write(ι_T)                     ▷ the same state as ι or just registers.
16             if rw_T = Read(ι) × Write(ι) or rw_T ⊆ {L_reg} × {L_reg} then
17                 regs ← {f | ℱ(f) = Reg and Write(ι, f) ≠ ∅}                      ▷ Any relevant
18                 E ← {Expr(p.f, ℳ) | f ∈ regs}                        ▷ source register can appear in
19                 H ← {f ↦ Field(ℱ_T(f), d, E) | f ∈ dom(ℱ_T)}                 ▷ a target field hole,
20                 I ← I ∪ {Expr((op_T, H), ℳ)}                  ▷ and any constant register or value.
21             end if
22         end for
23         body ← body · [Choose(I)]                              ▷ Append a hole over I to the body.
24     end for
25     return Expr((λp ∈ P(ι) . body), ℳ)                         ▷ A mini compiler sketch for ι.
26 end function
```

Figure 2.7: Pre-load sketch of length k and maximum depth d for ι, $\mathcal{A}_S$, $\mathcal{A}_T$, and $\mathcal{M}$. The $Load(E, \mathcal{A}_T, \mathcal{M})$ function returns a sequence of target instructions that load the immediate value described by the expression E into an unused scratch register; see Figure 2.5 for descriptions of other helper functions.

2.4.5 Solving Compiler Metasketches

JitSynth solves the metasketch CMS($\iota, \mathcal{A}_S, \mathcal{A}_T, \mathcal{M}$) by applying the host synthesizer to each of the generated sketches in turn until a mini compiler is found. If no mini compiler exists in the search space, this synthesis process runs forever. To check if a sketch $\mathcal{S}$ contains a mini compiler, JitSynth would ideally ask the host synthesizer to solve the following query, derived from Definitions 2.4–2.6:

$$\exists C \in \mathcal{S}.\ \forall \sigma_S \in \Sigma_S,\ \sigma_T \in \Sigma_T,\ p \in P(\iota). \sigma_S \cong_{\mathcal{M}} \sigma_T \Rightarrow \mathcal{A}_S(p, \sigma_S) \cong_{\mathcal{M}} \mathcal{A}_T(C(p), \sigma_T)$$

But recall that the state equivalence check $\cong_{\mathcal{M}}$ involves universally quantified formulas over memory addresses and register names. In principle, these innermost quantifiers are not problematic because they range over finite domains (bitvectors) so the formula remains decidable. In practice, however, they lead to intractable SMT queries. We therefore solve a stronger soundness query (Definition 2.11) that pulls these quantifiers out to obtain the standard $\exists\forall$ formula with a quantifier-free body. The resulting formula can be solved with CEGIS [SLTB+06], without requiring the underlying SMT solver to reason about quantifiers.

Definition 2.11 (Strongly Sound Mini Compiler) *Let $\mathcal{A}_S = (\mathcal{I}_S, \Sigma_S, \mathcal{T}_S, \Phi_S)$ and $\mathcal{A}_T = (\mathcal{I}_T, \Sigma_T, \mathcal{T}_T, \Phi_T)$ be two abstract register machines, $\cong_{\mathcal{M}}$ an injective state equivalence relation on their states Σ_S and Σ_T, and $C : P(\iota) \to List(P(\mathcal{I}_T))$ a function for some $\iota \in \mathcal{I}_S$. We say that C is a* strongly sound mini compiler *for $\iota_{\mathcal{M}}$ with respect to $\cong$ iff*

$$\forall \sigma_S \in \Sigma_S,\ \sigma_T \in \Sigma_T,\ p \in P(\iota),\ a \in dom(mem(\sigma_S)),\ r \in dom(reg(\sigma_S)).$$
$$\sigma_S \cong_{\mathcal{M},a,r} \sigma_T \Rightarrow \mathcal{A}_S(p, \sigma_S) \cong_{\mathcal{M},a,r} \mathcal{A}_T(C(p), \sigma_T)$$

where $\cong_{\mathcal{M},a,r}$ stands for the $\cong_{\mathcal{M}}$ formula with a and r as free variables.

The JitSynth synthesis procedure is sound and complete with respect to this stronger query (Theorem 2.2). The proof follows from the soundness and completeness of the host synthesizer, and the construction of the compiler metasketch. We discharge this proof using Lean theorem prover [dMKA+15].

Theorem 2.2 (Strong soundness and completeness of JitSynth) *Let $\mathcal{C} = CMS(\iota, \mathcal{A}_S, \mathcal{A}_T, \mathcal{M})$ be the compiler metasketch for the abstract instruction ι, machines $\mathcal{A}_S$ and $\mathcal{A}_T$, and the state mapping $\mathcal{M}$. If JitSynth terminates and returns a program C when applied to $\mathcal{C}$, then C is a strongly sound mini compiler for ι and $\mathcal{A}_T$ (soundness). If there is a strongly sound mini compiler in the most general search space $\{\textsc{Naive}(k, d, \iota, \mathcal{A}_S, \mathcal{A}_T, \mathcal{M}) \mid k, d \in \mathbb{N}\}$, then JitSynth will terminate on $\mathcal{C}$ and produce a program (completeness).*

2.5 Implementation

We implemented JitSynth as described in Section 2.2 using Rosette [TB14] as our host language. Since the search spaces for different compiler lengths are disjoint, the JitSynth implementation searches these spaces in parallel [BTGC16]. We use $\Phi(\boldsymbol{p}) = \texttt{length}(\boldsymbol{p})$ as the fuel function for all languages studied in this work. This provides sufficient fuel for evaluating programs in these languages that are accepted by the OS kernel. For example, the Linux kernel requires eBPF programs to be loop-free, and it enforces this restriction with a conservative static check; programs that fail the check are not passed to the JIT [GAG+19].

Synthesizing Common Sequences. In addition to separately synthesizing pre-loads for parameters, JitSynth also synthesizes a few other small sequences used commonly in compiled programs.

One such class of sequences are register extensions. The target abstract register machine may have larger register values than the source machine. In these cases, the compiler may often need to either sign-extend the target register values, filling the upper bits of the register with the sign bit, or zero-extend, filling the upper bits of the register with 0. JITSYNTH synthesizes a sequence for each of these extension types. When $L_{reg} \in Read(\iota)$ of source abstract instruction ι, JITSYNTH will try sketches that extend used registers in both ways.

Additionally, JITSYNTH synthesizes sequences that both load the PC into a temporary register and write to the PC from a register value. These operations are needed for compiling jumps, and are reused in many mini compilers.

NOP Padding and Removal. JITSYNTH synthesizes mini compilers of equal length to ensure that the target PC for jumps can be computed as a multiple of the source PC. To do so, JITSYNTH pads instruction sequences with NOPs to match the length of the largest mini compiler. To mitigate the performance impact this incurs, we implemented a trusted compiler pass that removes the NOP instructions while preserving the correctness of the compiled code.

2.6 Evaluation

This section evaluates JITSYNTH by answering the following research questions:

RQ1: Can JITSYNTH synthesize correct and performant compilers for real-world source and target languages?

RQ2: How effective are the sketch optimizations described in Section 2.4?

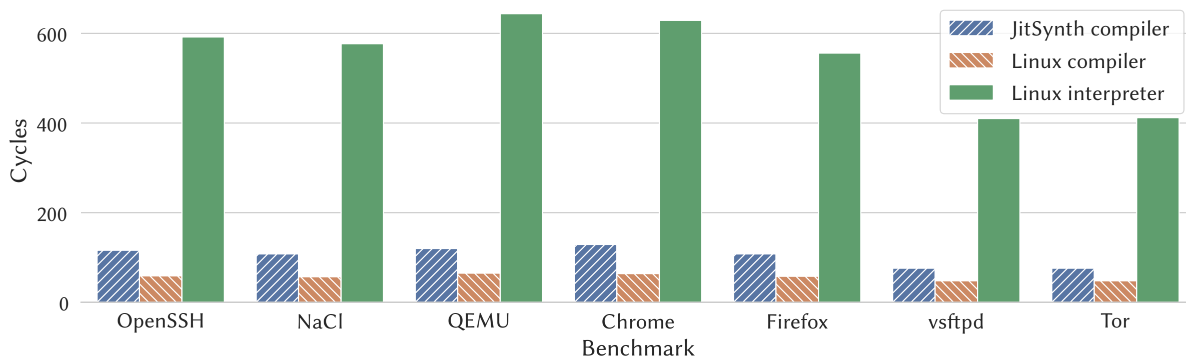


Figure 2.8: Execution time of eBPF benchmarks on the HiFive Unleashed RISC-V development board, using the existing Linux eBPF to RISC-V compiler, the JITSYNTH compiler, and the Linux eBPF interpreter. Measured in processor cycles.

2.6.1 Synthesizing compilers for real-world source-target pairs

To demonstrate the effectiveness of JITSYNTH, we applied JITSYNTH to synthesize compilers for three different source-target pairs: eBPF to 64-bit RISC-V, classic BPF to eBPF, and libseccomp to eBPF. This subsection describes our results for each of the synthesized compilers.

eBPF to RISC-V. As a case study, we applied JITSYNTH to synthesize a compiler from eBPF to 64-bit RISC-V. It supports 87 of the 102 eBPF instruction opcodes; unsupported eBPF instructions include function calls, endianness operations, and atomic instructions. To validate that the synthesized compiler is correct, we ran the existing eBPF test cases from the Linux kernel; our compiler passes all test cases it supports. In addition, our compiler avoids bugs previously found in the existing Linux eBPF-to-RISC-V compiler in Linux [Nel19]. To evaluate performance, we compared against the existing Linux compiler. We used the same set of benchmarks used by Jitk [WLZ+14], which includes system call filters from widely used applications. Because these benchmarks were originally for classic BPF, we first compile them to eBPF using the existing Linux classic-BPF-to-eBPF compiler as a preprocessing step. To run the benchmarks, we execute

the generated code on the HiFive Unleashed RISC-V development board [SiF18], measuring the number of cycles. As input to the filter, we use a system call number that is allowed by the filter to represent the common case execution.

Figure 2.8 shows the results of the performance evaluation. eBPF programs compiled by JitSynth JIT compilers show an average slowdown of 1.82× compared to programs compiled by the existing Linux compiler. This overhead results from additional complexity in the compiled eBPF jump instructions. Linux compilers avoid this complexity by leveraging bounds on the size of eBPF jump offsets. JitSynth-compiled programs get an average speedup of 5.24× compared to interpreting the eBPF programs. This evidence shows that JitSynth can synthesize a compiler that outperforms the current Linux eBPF interpreter, and nears the performance of the Linux compiler, while avoiding bugs. We acknowledge the gap that still remains between the synthesized compiler and the Linux compiler; we leave further narrowing this gap for future work.

Classic BPF to eBPF. Classic BPF is the original, simpler version of BPF used for packet filtering which was later extended to eBPF in Linux. Since many applications still use classic BPF, Linux must first compile classic BPF to eBPF as an intermediary step before compiling to machine instructions. As a second case study, we used JitSynth to synthesize a compiler from classic BPF to eBPF. Our synthesized compiler supports all classic BPF opcodes. To evaluate performance, we compare against the existing Linux classic-BPF-to-eBPF compiler. Similar to the RISC-V benchmarks, we run each eBPF program with input that is allowed by the filter. Because eBPF does not run directly on hardware, we measure the number of instructions executed instead of processor cycles.

Figure 2.9 shows the performance results. Classic BPF programs generated by JitSynth compilers execute an average of 2.28× more instructions than those compiled by Linux.

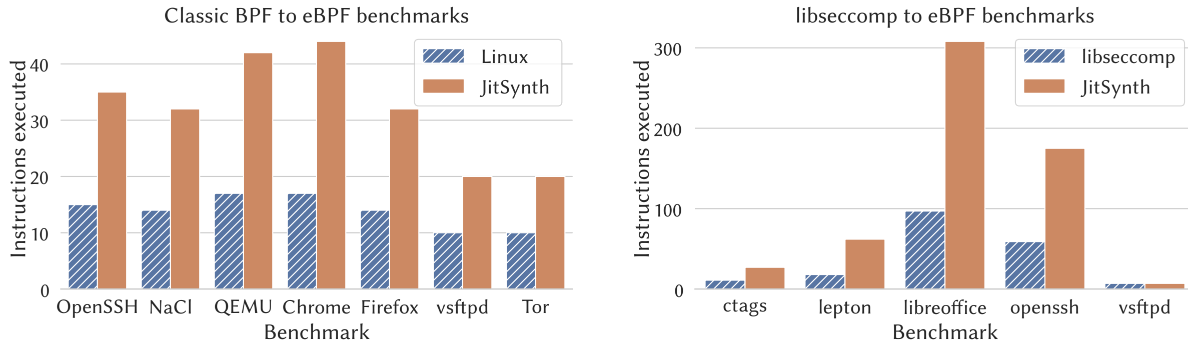


Figure 2.9: Performance of code generated by JitSynth compilers compared to existing compilers for the classic BPF to eBPF benchmarks (left) and the libseccomp to eBPF benchmarks (right). Measured in number of instructions executed.

libseccomp to eBPF. libseccomp is a library used to simplify construction of BPF system call filters. The existing libseccomp implementation compiles to classic BPF; we instead choose to compile to eBPF because classic BPF has only two registers, which does not satisfy the assumptions of JitSynth. Since libseccomp is a library and does not have distinct instructions, libseccomp itself does not meet the definition of an abstract register machine; we instead introduce an intermediate libseccomp language which does satisfy this definition. Our full libseccomp to eBPF compiler is composed of both a trusted program to translate from libseccomp to our intermediate language and a synthesized compiler from our intermediate language to eBPF.

To evaluate performance, we select a set of benchmark filters from real-world applications that use libseccomp, and measure the number of eBPF instructions executed for an input the filter allows. Because no existing compiler exists from libseccomp to eBPF directly, we compare against the composition of the existing libseccomp-to-classic-BPF and classic-BPF-to-eBPF compilers.

Figure 2.9 shows the performance results. libseccomp programs generated by JitSynth execute 2.61× more instructions on average compared to the existing libseccomp-to-eBPF compiler stack. However, the synthesized compiler avoids bugs previously found in the libseccomp-to-classic-BPF

compiler [Hor19].

2.6.2 Effectiveness of sketch optimizations

In order to evaluate the effectiveness of the search optimizations described in Section 2.4, we measured the time JitSynth takes to synthesize each of the three compilers with different optimizations enabled. Specifically, we run JitSynth in three different configurations: (1) using Naive sketches, (2) using RW sketches, and (3) using PLD sketches. For each configuration, we ran JitSynth with a timeout of 48 hours (or until out of memory). Figure 2.10 shows the time to synthesize each compiler under each configuration. Note that these figures do not include time spent computing read and write sets, which takes less than 11 minutes for all cases. Our results were collected using an 8-core AMD Ryzen 7 1700 CPU with 16 GB memory, running Racket v7.4 and the Boolector [NPB15] solver v3.0.1-pre.

When synthesizing the eBPF-to-RISC-V compiler, JitSynth runs out of memory with Naive sketches, reaches the timeout with RW sketches, and completes synthesis with PLD sketches. For the classic-BPF-to-eBPF compiler, JitSynth times out with both Naive sketches and RW sketches. JitSynth only finishes synthesis with PLD sketches. For the libseccomp-to-eBPF compiler, all configurations finish, but JitSynth finishes synthesis about 34× times faster with PLD sketches than with Naive sketches. These results demonstrate that the techniques JitSynth uses are essential to the scalability of JIT synthesis.

2.7 Related Work

JIT compilers for in-kernel languages. JIT compilers have been widely used to improve the extensibility and performance of systems software, such as OS kernels [CCK+13, Eng96, Fle17, Myr11]. One notable system is Jitk [WLZ+14]. It builds on the CompCert compiler [Ler09] to

Compiler	NAIVE sketch	RW sketch	PLD sketch
eBPF to RISC-V	X	X	44.4h
classic BPF to eBPF	X	X	1.2h
libseccomp to eBPF	4.0h	43.5m	7.1m

Figure 2.10: Synthesis time for each source-target pair, broken down by set of optimizations used in the sketch. An X indicates that synthesis either timed out or ran out of memory.

compile classic BPF programs to machine instructions. Both Jitk and CompCert are formally verified for correctness using the Coq interactive theorem prover. Jitk is further extended to support eBPF [Sob15]. Like Jitk, JITSYNTH provides formal correctness guarantees of JIT compilers. Unlike Jitk, JITSYNTH does not require developers to write either the implementation or proof of a JIT compiler. Instead, it takes as input interpreters of both source and target languages and state-mapping functions, using automated verification and synthesis to produce a JIT compiler.

An in-kernel extension system such as eBPF also contains a *verifier*, which checks for safety and termination of input programs [GAG+19, WLZ+14]. JITSYNTH assumes a well-formed input program that passes the verifier and focuses on the correctness of JIT compilation.

Synthesis-aided compilers. There is a rich literature that explores generating and synthesizing peephole optimizers and superoptimizers based on a given ISA or language specification [BA06, DF84, GJTV11, JNR02, Mas87, SCC+17, SSA13]. Bansal and Aiken described a PowerPC-to-x86 binary translator using peephole superoptimization [BA08]. Chlorophyll [PJS+14] applied synthesis to a number of compilation tasks for the GreenArrays GA144 architecture, including code partitioning, layout, and generation. JITSYNTH bears the similarity of translation between a source-target pair of languages and shares the challenge of scaling up synthesis. Unlike existing work, JITSYNTH synthesizes a *compiler* written in a host language, and uses compiler metasketches for efficient synthesis.

Compiler testing. Compilers are complex pieces of software and are known to be difficult to get right [MTDC19]. Recent advances in compiler testing, such as Csmith [YCER11] and EMI [ZSS17], have found hundreds of bugs in GCC and LLVM compilers. Alive [LHL19, LMNR15] and Serval [NBG+19] use automated verification techniques to uncover bugs in the LLVM's peephole optimizer and the Linux kernel's eBPF JIT compilers, respectively. JitSynth complements these tools by providing a correctness-by-construction approach for writing JIT compilers.

2.8 Conclusion

This chapter presents a new technique for synthesizing JIT compilers for in-kernel DSLs. The technique creates per-instruction compilers, or compilers that independently translate single source instructions to sequences of target instructions. In order to synthesize each per-instruction compiler, we frame the problem as search using compiler metasketches, which are optimized using both read and write set information as well as pre-synthesized load operations. We implement these techniques in JitSynth and evaluate JitSynth over three source and target pairs from the Linux kernel. Our evaluation shows that (1) JitSynth can synthesize correct and reasonably performant compilers for real in-kernel languages, and (2) the optimizations discussed in this chapter make the synthesis of these compilers tractable to JitSynth. As future in-kernel DSLs are created, JitSynth can reduce both the programming and proof burden on developers writing compilers for those DSLs. The JitSynth source code is publicly available at `https://github.com/uw-unsat/jitsynth`.

Chapter 3

DepSynth: Automatically Developing Crash Consistency Mechanisms

Reliable storage systems must be *crash consistent*—guaranteed to recover to a consistent state after a crash. Crash consistency is non-trivial as it requires maintaining complex invariants about persistent data structures in the presence of caching, reordering, and system failures. Current programming models offer little support for implementing crash consistency, forcing storage system developers to roll their own consistency mechanisms. Bugs in these mechanisms can lead to severe data loss for applications that rely on persistent storage.

This chapter presents a new *synthesis-aided* programming model for building crash-consistent storage systems. In this approach, storage systems can assume an *angelic crash-consistency* model, where the underlying storage stack promises to resolve crashes in favor of consistency whenever possible. To realize this model, we introduce a new *labeled writes* interface for developers to identify their writes to disk, and develop a program synthesis tool, DepSynth, that generates *dependency rules* to enforce crash consistency over these labeled writes. We evaluate our model in a case study on a production storage system at Amazon Web Services. We find that DepSynth

can automate crash consistency for this complex storage system, with similar results to existing expert-written code, and can automatically identify and correct consistency and performance issues.

3.1 Overview

Many applications build on storage systems such as file systems and key-value stores to reliably persist user data even in the face of full-system crashes (e.g., power failures). Guaranteeing this reliability requires the storage system to be *crash consistent*: after a crash, the system should recover to a consistent state without losing previously persisted data. The state of a storage system is consistent if it satisfies the representation invariants of the underlying persistent data structures (e.g., a free data block must not be linked by any file's inode). Crash consistency is notoriously difficult to get right [YTEM06, PCA+14, ZTH+14], due to performance optimizations in modern software and hardware that can reorder writes to disk or hold pending disk writes in a volatile cache. In normal operation, these optimizations are invisible to the user, but a crash can expose their partial effects, leading to inconsistent states.

A number of general-purpose approaches exist to implement crash consistency, including journaling [PAA05], copy-on-write data structures [RBM13], and soft updates [GP94]. However, implementing a storage system using these approaches is still challenging for two reasons. First, practical storage systems combine crash consistency techniques with optimizations such as log-bypass writes and transaction batching to improve performance [Twe98]. These optimizations and their interactions are subtle, and have led to severe crash-consistency bugs in well-tested storage systems [LADADL13, CCK+17]. Second, developers must implement their system using low-level APIs provided by storage hardware and kernel I/O stacks, which offer no direct support for enforcing consistency properties. Instead they provide only durability primitives such as

flushes, and require the developer to roll their own consistency mechanisms on top of them. While prior work offers testing [MMP+18, YTEM06] and verification [CZC+15, SBTW16] tools for validating crash consistency, these tools do not alleviate the burden of implementing crash-consistent systems.

This chapter presents a new synthesis-aided programming model for building crash-consistent storage systems. The programming model consists of three parts: a high-level storage interface based on *labeled writes*; a synthesis engine for turning labeled writes and a desired crash consistency property into a set of *dependency rules* that writes to disk must respect; and a *dependency-aware buffer cache* that enforces the synthesized rules at run time. Together, these three components let developers keep their implementation free of hardcoded optimizations and mechanisms for enforcing consistency. Instead, developers can focus on the key aspects of their storage system—functional correctness, crash consistency, and performance—one at a time. Their development workflow consists of three steps.

First, developers implement their system against a higher-level storage interface by providing *labels* for each write their system makes to disk. Labels provide information about the data structure the write targets and the context for the write (e.g., the transaction it is part of). For example, a simple journaling file system might require two writes to append to the journal: one to append the data block to the tail of the journal (labeled data) and one to update a superblock that records a pointer to that tail (labeled superblock). This higher-level interface allows the developer to assume a stronger *angelic nondeterminism* model for crashes—the system promises that crash states will *always* satisfy the developer's crash consistency property if possible—simplifying the implementation effort.

Second, to make their implementation crash consistent even on relaxed storage stacks, the developer uses a new program synthesizer, DEPSYNTH, to automatically generate *dependency rules* that writes to disk must respect. A dependency rule uses labels to define an ordering requirement between two writes: writes with one label must be persisted on disk before corresponding

writes with the second label. The DEPSYNTH synthesizer takes three inputs: the storage system implementation, a desired *crash consistency predicate* for disk states of the storage system (i.e., a representation invariant for on-disk data structures), and a collection of small *litmus test* programs [AMSS11, BKL+16] that exercise the storage system. Given these inputs, DEPSYNTH searches a space of happens-before graphs to automatically generate a set of dependency rules that guarantee the crash-consistency predicate for every litmus test. Although this approach is example-guided and so only guarantees crash consistency on the supplied tests, the dependency rule language is constrained to make it difficult to overfit to the tests, and so in practice the rules generalize to arbitrary executions of the storage system.

Third, developers run their storage system on top of a *dependency-aware buffer cache* that enforces the synthesized dependency rules. For example, in a journaling file system, the superblock pointer to the tail of the journal must never refer to uninitialized data. DEPSYNTH will synthesize a dependency rule enforcing this consistency predicate by saying that data writes must happen before superblock writes. At run time, the dependency-aware buffer cache enforces this rule by delaying sending writes labeled superblock to disk until the corresponding data write has persisted. The dependency-aware buffer cache is free to reorder writes in any way to achieve good performance on the underlying hardware (e.g., by scheduling around disk head movement or SSD garbage collection) as long as it respects the dependency rules.

We evaluate the effectiveness and utility of DEPSYNTH in a case study that applies it to ShardStore [BJA+21], a production key-value store used by the Amazon S3 object storage service. We show that DEPSYNTH can rapidly synthesize dependency rules for this storage system. By comparing those rules to the key-value store's existing crash-consistency behavior, we find that DEPSYNTH achieves similar results to rules hand-written by experts, and even corrects an existing crash-consistency issue in the system automatically. We also show that dependency rules synthesized by DEPSYNTH generalize beyond the example litmus tests used for synthesis, and that

DepSynth can be used for storage systems beyond key-value stores.

In summary, this work makes three contributions:

- A new programming model for building storage systems that automates the implementation of crash consistency guarantees;
- DepSynth, a synthesis tool that can infer the dependency rules sufficient for a storage system to be crash consistent; and
- An evaluation showing that DepSynth supports different storage system designs and scales to production-quality systems.

The remainder of this chapter is organized as follows. Section 3.2 gives a walk-through of building a simple storage system with DepSynth. Section 3.3 defines the DepSynth programming model, including labeled writes and dependency rules. Section 3.4 describes the DepSynth synthesis algorithm for inferring dependency rules, and Section 3.5 details DepSynth's implementation in Rosette. Section 3.6 evaluates the effectiveness of DepSynth. Section 3.7 discusses related work, and Section 3.8 concludes.

3.2 DepSynth by Example

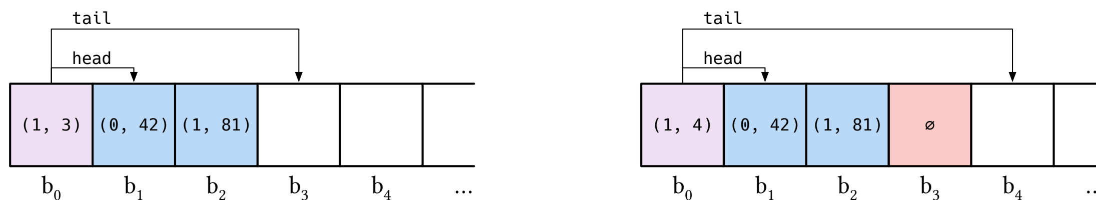
This section illustrates the DepSynth development workflow by walking through the implementation of a simple storage system. We show how a developer can build a storage system with labeled writes while assuming a strong crash consistency model, and use DepSynth to automatically make that system crash consistent on real storage stacks.

Log-structured storage systems. A log-structured storage system persists user data in a sequential log on disk [RO91]. This design forsakes complex on-disk data structures in favor of

one with simple invariants and, as a result, simpler crash consistency requirements. However, although log-structured storage systems are well studied, their precise consistency requirements can be subtle in the face of the caching and reordering optimizations used by the modern storage stack.

Consider implementing a simple key-value store as a log-structured storage system. The on-disk data structure comprises two parts as shown in Fig. 3.1a: a log that stores key-value pairs (with one pair per block), and a superblock that holds pointers to the head and tail of the log. We will assume that single-block writes (`disk.write`) are atomic, that each key-value pair fits in one block, and that the log does not run out of space. To implement this system, the developer writes `put` and `get` methods that interact with the disk:

```
class KeyValueStore(DepSynth):
  def __init__(self):
    self.superblock = disk.read(0)
    if self.superblock.empty():  # initialize an empty disk
      self.superblock_head, self.superblock_tail = 1, 1
    else:
      self.superblock_head, self.superblock_tail = from_block(superblock)
    self.epoch = 0

  def put(self, key: int, value: int):
    address = self.superblock_tail
    self.superblock_tail += 1

    new_block = to_block(key, value)
    disk.write(address, new_block, ("log", self.epoch))

    new_superblock = to_block(self.superblock_head, self.superblock_tail)
    disk.write(0, new_superblock, ("superblock", self.epoch))

    self.epoch += 1

  def get(self, key: int) -> Optional[int]:
    address = self.superblock_tail - 1
    while address >= self.superblock_head:
      block = disk.read(address)
      current_key, current_value = from_block(block)
      if current_key == key:
        return current_value
      address -= 1
    return None
```



(a) On-disk layout of a simple log-structured key-value store. Each block holds a (key, value) pair. The first block is a superblock that holds pointers to the head and tail of the log.

(b) Possible on-disk state after a crash, leaving the superblock pointing to a range that includes an invalid block.

Figure 3.1: The on-disk layout of a simple key-value store. Arrows denote pointers and boxes are blocks.

Calls to `disk.read` and `disk.write` illustrate our new higher-level storage interface: `disk.read` is unchanged from the usual system call, taking as input an address on the disk to read from; and `disk.write` takes as input an address on the disk to write to, the block data to write to that address, and a third *label* argument. A label is a pair of a string *name* and an integer *epoch*. Labels serve as identities for writes: the name describes the data structure the write targets, while the epoch relates writes across different data structures. This implementation uses the name part of the label to distinguish writes of new log blocks and writes to the superblock,[1] and uses the epoch part as a logical clock that relates the two writes generated by a single `put` call. Labels exist only in memory while a write is in-flight, and are never persisted to disk.

While this implementation is functionally correct, it would not be crash consistent if implemented on a classical storage stack. The issue is with the ordering of log and superblock writes: even though the code suggests that the superblock write comes after the log write, optimizations in the storage stack could reorder the two writes and lead to a crash state where the superblock is updated but its corresponding new log block is not, as Fig. 3.1b shows. This would leave the

[1]For this system we could distinguish the two data structures without labels—superblock writes are to address 0 while log writes are to non-zero addresses—but in general, storage systems reuse addresses over time and so this mapping is not static.

`superblock_tail` pointer referring to an uninitialized disk block. What we need for consistency is a way to preclude this reordering. One solution in the DepSynth programming model would be for the developer to manually implement a *dependency rule* that prevents this reordering:

```
def __init__(self):
  self.rule("superblock", "log", eq)
```

A dependency rule `rule("a", "b", eq)` specifies an ordering constraint: a write labeled with name `"a"` must not be sent to disk until after a write labeled with name `"b"`. We say that such a rule means write `"a"` *depends on* write `"b"`, or equivalently that write `"b"` must *happen before* write `"a"`. The third argument to `rule` is an *epoch predicate* that scopes the rule using the epoch in each label. Here, the `eq` predicate restricts the rule to only apply to pairs of writes whose labels have equal epochs. This rule means that superblock updates cannot be persisted on disk until a log block write with the same epoch is persisted first, ruling out the reordering behavior that could make the log inconsistent.

Dependency rule synthesis. While the developer could specify the above dependency rule manually, our programming model does not require them to, and distilling the correct set of rules for a complex storage system is difficult to do by hand. The challenge is a semantic gap: the developer's desired high-level consistency property is about the on-disk data structure as a whole, but the implementation of consistency can only refer to individual block-sized writes. We bridge this gap with DepSynth, a program synthesis tool that can *automatically infer* the dependency rules sufficent to make a storage system crash consistent.

DepSynth takes three inputs. First, it takes as input the implementation of the storage system. Second, it takes as input a crash consistency predicate, written as an executable checker over a disk state. The crash consistency predicate defines the property that should be true of *every* state of the disk, including after crashes. For our log-structured key-value store, our desired consistency

property is that the `superblock_tail` pointer never gets ahead of the blocks that have been written to the log. We can implement this property by checking that all blocks in the log are valid log blocks (we omit an implementation of `valid` for brevity, but it could validate a checksum of the block):

```
def consistent(self) -> bool:
  ret = True
  for address in range(self.superblock_head, self.superblock_tail):
    block = disk.read(address)
    ret = ret and valid(block)
  return ret
```

Finally, DepSynth takes as input a collection of *litmus tests*, small programs that exercise the storage system. Litmus tests are widely used to communicate the semantics of memory consistency models [AMSS11, WBSC17], and have also been used to communicate crash consistency models [BKL+16]. A DepSynth litmus test comprises two executable programs *initial* and *main*. Both programs take as input a reference to the storage system. The *initial* program sets up some initial state in the system, and cannot crash. The *main* program manipulates the system state, and can crash at any point. For example, this is a simple litmus test that starts from a single log entry and appends two more:

```
class SingleEntry_TwoAppend(LitmusTest):
  def initial(self, store: KeyValueStore):
    store.put(0, 42)

  def main(self, store: KeyValueStore):
    store.put(1, 81)
    store.put(2, 37)
```

As with previous work on memory consistency models [AMSS11, BT17], the developer can draw litmus tests from a number of sources: they may be hand-written by the developer, drawn from a common set of tests for important properties, generated automatically by a fuzzer or program enumerator, or intelligently generated by analyzing the on-disk data structures used by the storage

system [AMSS10].

Given these three inputs, DEPSYNTH automatically synthesizes a set of dependency rules that suffice to guarantee the crash-consistency predicate holds on all crash states generated by all litmus tests. For our example log-structured key-value store, DEPSYNTH synthesizes two dependency rules:

```
def __init__(self):
  self.rule("superblock", "log", eq)
  self.rule("superblock", "superblock", gt)
```

The first rule is the same rule we hand-wrote earlier. The second rule fixes a subtle crash-consistency bug in our hand-written implementation: while the first rule ensures consistency for a *single* put operation, it still allows `superblock_tail` to get ahead of the log if writes from *multiple* puts are reordered with each other (for example, reordering writes from the first and second puts in the litmus test above). The second rule prevents this reordering using the `gt` epoch predicate, which specifies that a superblock write with epoch i cannot be persisted to disk until all superblock writes with lower epochs $j < i$ are persisted first. The combination of these rules precludes the problematic reordering and guarantees that the superblock always refers to a valid *range* of log blocks, rather than only requiring the block at `superblock_tail` to be valid.

3.3 Reasoning About Crash Consistency

The DEPSYNTH workflow includes a new high-level interface for building storage systems and a synthesis tool for automatically making those systems crash consistent. This section describes the high-level interface, including labeled writes and dependency rules, and presents a logical encoding for reasoning about crashes of systems that use this interface. Section 3.4 then presents the DEPSYNTH synthesis algorithm for inferring sufficient dependency rules to make a storage

system crash consistent.

3.3.1 Disk Model and Dependency Rules

In the DepSynth programming model, storage systems run on top of a disk model d that provides two operations:

- $d.\texttt{write}(a, v, l)$: write a data block v to disk address a with label l
- $d.\texttt{read}(a)$: read a data block at disk address a

We assume that single-block write operations are atomic, as in previous work [SBTW16, CZC⁺15]. These interfaces are similar to the standard POSIX `pwrite` and `pread` APIs, except that the `write` operation additionally takes as input a *label* for the write. A label $l = \langle n, t \rangle$ is a pair of a *name* string n and an *epoch* integer t. Labels allow the developer to provide identities for each write their system performs, which dependency rules (described below) can inspect to enforce ordering requirements. Although the two components of a label together identify a write, developers use them for separate purposes: the name indicates which on-disk data structure the write targets, while the epoch associates related writes with different names. Names are strings but are not interpreted by our workflow other than to check equality between them. Epochs are integers that dependency rules use as logical clocks to impose orderings on related writes.

Dependency rules. DepSynth synthesizes declarative *dependency rules* to enforce consistency requirements for a storage system that uses labeled writes.

Definition 3.1 (Dependency rule) *A* dependency rule $n_1 \rightsquigarrow_p n_2$ *comprises two names* n_1 *and* n_2 *and an epoch predicate* $p(t_1, t_2)$ *over pairs of epochs. Given two labels* $l_a = \langle n_a, t_a \rangle$ *and* $l_b = \langle n_b, t_b \rangle$*, we say that a dependency rule* $n_1 \rightsquigarrow_p n_2$ matches l_a *and* l_b *if* $n_a = n_1$*,* $n_b = n_2$*, and* $p(t_a, t_b)$ *is true.*

Dependency rules define ordering requirements over all writes with labels that match them, and the dependency-aware buffer cache enforces these rules at run time. More precisely, the dependency-aware buffer cache enforces *dependency safety* for all writes it sends to disk:

Definition 3.2 (Dependency safety) *A dependency-aware buffer cache maintains* dependency safety *for a set of dependency rules R if, whenever a storage system issues two writes* $d.\texttt{write}(a_1, s_1, l_1)$ *and* $d.\texttt{write}(a_2, s_2, l_2)$*, and a rule* $n_a \rightsquigarrow_p n_b \in R$ *matches* l_1 *and* l_2*, then the cache ensures the write to* a_1 *does not persist until the write to* a_2 *is persisted on disk.*

In other words, all crash states of the disk that include the effect of the first write must also include the effect of the second write. Section 3.3.3 will specify dependency safety more formally by defining the crash behavior of a disk in first-order logic.

The epoch predicate of a dependency rule reduces the scope of the rule to only apply to some writes labeled with the relevant names. Given two labels $l_1 = \langle n_1, t_1 \rangle$ and $l_2 = \langle n_2, t_2 \rangle$, a dependency rule $n_1 \rightsquigarrow_p n_2$ can use one of three epoch predicates: =, >, and <, which restrict the rule to apply only when $t_1 = t_2$, $t_1 > t_2$, and $t_1 < t_2$, respectively. These variations allow dependency rules to specify ordering requirements over unbounded executions of the storage system without adding unnecessary dependencies between *all* operations with certain names.

Together, the name and epoch components of labels allow dependency rules to define a variety of important consistency requirements, depending on how the developer chooses to label their writes. For example, if all writes generated by a related operation (e.g., a top-level API operation like `put` in a key-value store) share the same epoch t, then rules using the = epoch predicate can impose consistency requirements on individual operations, such as providing transactional semantics. As another example, rules using the > epoch predicate can be used as barriers for all previous writes, and so can help to implement operations like garbage collection that manipulate an entire data structure.

Dependency-aware buffer cache. At run time, storage systems implemented with the DepSynth programming model execute on top of a *dependency-aware buffer cache*. This buffer cache is configured with a set of dependency rules at initialization time, and enforces those rules on all writes executed by the storage system.

The dependency-aware buffer cache is inspired by previous higher-level storage APIs such as those used by Featherstitch [FMK+07] and ShardStore [BJA+21], which also provide interfaces for specifying ordering requirements for writes. Both of these interfaces are imperative: they require the developer to manually construct a dependency graph for each write they execute, and so closely intertwine the ordering requirements with the implementation, as constructing these graphs requires sharing graph nodes (*patchgroups* in Featherstitch and *dependencies* in ShardStore) across threads and operations. In contrast, the dependency-aware buffer cache interface is declarative: the dependency rules are configured once, and then automatically applied to all relevant writes without requiring the developer to manually construct graphs or invoke consistency primitives like `fsync`.

The implementation details of the dependency-aware buffer cache are outside the scope of this chapter and follow the examples of Featherstitch and ShardStore. An implementation could use a variety of consistency and durability primitives provided by disks, including force-unit-access writes, cache flush commands, or ordering barriers. We trust the correctness of the dependency-aware buffer cache, and specifically we assume it enforces dependency safety (Definition 3.2).

3.3.2 Storage Systems and Litmus Tests

To apply DepSynth, developers provide three inputs: a storage system implementation, a collection of litmus tests that exercise the storage system, and a crash consistency predicate for the system.

Storage system implementations. Developers implement a storage system for DepSynth by defining a collection of API operations $\mathcal{O}$ and an implementation function for each operation:

Definition 3.3 (Storage system implementation) *A storage system implementation $\mathcal{O} = \{O_a, O_b, \dots\}$ is a set of API operations O_i and, for each O_i, an* implementation function *$I_{O_i}(d, \boldsymbol{x})$ that takes as input a disk state d and a vector of other inputs $\boldsymbol{x}$ and issues write operations to mutate disk d.*

DepSynth requires implementation functions to support being symbolically evaluated with respect to a symbolic disk state d. In this work, we use Rosette [TB14] as our symbolic evaluator; this allows implementation functions to be written in Racket and automatically lifted to support the necessary symbolic evaluation, so long as their executions are deterministic and bounded.

We say that a *program P* is a sequence of calls $[O_1(\boldsymbol{x}_1), \dots, O_n(\boldsymbol{x}_n)]$ to API operations $O_i \in \mathcal{O}$. Given a program P, we write $Evaluate_{\mathcal{O}}(P)$ for the function that symbolically evaluates each $I_{O_i}(d, \boldsymbol{x}_i)$ in turn, starting from a symbolic disk d, and returns a *trace* of labeled write operations $[w_1, \dots, w_n]$ that the program performed. The trace does not need to include read operations as they cannot participate in ordering requirements.

Litmus tests. DepSynth synthesizes dependency rules from a set of example *litmus tests*, which are small programs that exercise the storage system and demonstrate its desired consistency behavior. A litmus test $T = \langle P_{initial}, P_{main} \rangle$ is a pair of programs that each invoke operations of the storage system. The initial program $P_{initial}$ sets up an initial state of the storage system by, for example, prepopulating the disk with files or objects. It will be executed starting from an empty disk, and cannot crash. The main program P_{main} then tests the behavior of the storage system starting from that initial state. DepSynth will exercise all possible crash states of the main program.

Litmus tests are widely used to communicate the semantics of memory consistency models to developers [AMSS11, WBSC17], and have also been used to communicate crash consis-

tency [BKL+16] and to search for crash consistency bugs in storage systems [MMP+18]. DepSynth is agnostic to the source of the litmus tests it uses so long as they fit the definition of a program (i.e., are straight-line and deterministic).

Crash consistency predicates. To define crash consistency for their system, developers also provide a *crash-consistency predicate Consistent*(d) that takes a disk state d and returns whether the disk state should be considered consistent. The crash-consistency predicate should include representation invariants for the storage system's on-disk data structures. For example, a file system like ext2 might require that all block pointers in inodes refer to blocks that are allocated (i.e., no dangling pointers). These properties correspond to those that can be checked by an `fsck`-like checker [Hen07]. The crash-consistency predicate can also include stronger properties such as checking the atomic-replace-via-rename property for POSIX file systems [BKL+16, PCA+14].

3.3.3 Reasoning About Crashes

To reason about the crash behaviors of a storage system, we encode the semantics of dependency rules and litmus tests in first-order logic based on existing work on storage verification [SBTW16]. We first encode the behavior of a single write operation, and then extend that encoding to executions of entire programs.

Write operations. We model the behavior of a disk write operation as a transition function $f_{\texttt{write}}(d, a, v, s)$, that takes four inputs: the current disk state d, the disk address a to write to, the new block value v to write, and a *crash flag* s, a boolean that is used to encode the effect of a crash on the resulting disk state. Given these inputs, $f_{\texttt{write}}$ returns the resulting disk state after applying

the operation. The effect of a write operation is visible on the disk only if s is true:

$$f_{\texttt{write}}(d, a, v, s) = d[a \mapsto \texttt{if}\ s\ \texttt{then}\ v\ \texttt{else}\ d(a)].$$

Program executions. Given the trace of write operations $[w_1, \ldots, w_n] = \mathit{Evaluate}_O(P)$ executed by a program P against storage system O, and for each write its corresponding crash flag s_i, we can define the final disk state of the program by just applying the transition function in sequence:

$$\mathit{Run}\,([\texttt{write}(a_1, v_1, l_1), w_2, \ldots, w_n], [s_1, \ldots, s_n], d) = \mathit{Run}\,([w_2, \ldots, w_n], [s_2, \ldots, s_n], f_{\texttt{write}}(d, a_1, v_1, s_1))$$
$$\mathit{Run}\,([], [], d) = d$$

We call the vector $\mathbf{s} = [s_1, \ldots, s_n]$ of crash flags for each operation in the trace a *crash schedule.*

Not all crash schedules are possible. At run time, the dependency-aware buffer cache constrains the set of *valid crash schedules* by applying the dependency rules it is configured with:

Definition 3.4 (Valid crash schedule) *Let* $[w_1, \ldots, w_n] = \mathrm{Evaluate}_O(P)$ *be the trace of operations executed by a program* P *on storage system* O, R *be a set of dependency rules, and* $\mathbf{s} = [s_1, \ldots, s_n]$ *the crash schedule for the trace. The crash schedule* $\mathbf{s}$ *is* valid *for the program* P *and set of rules* R*, written* $\mathrm{Valid}_R(\mathbf{s}, P)$*, if for all operations* $w_i = \texttt{write}(a_i, v_i, l_i)$ *and* $w_j = \texttt{write}(a_j, v_j, l_j)$*, whenever there exists a rule* $n_a \rightsquigarrow_p n_b \in R$ *that matches* l_i *and* l_j*, then* $s_i \longrightarrow s_j$*.*

This definition is a logical encoding of dependency safety (Definition 3.2): if $s_i \longrightarrow s_j$, then write w_j is guaranteed to be persisted on disk whenever write w_i is.

Finally, we can define crash consistency for a litmus test $T = \langle P_{initial}, P_{main} \rangle$ as a function of a set of dependency rules R:

Definition 3.5 (Single-test crash consistency) *Let $T = \langle P_{\text{initial}}, P_{\text{main}} \rangle$ be a litmus test. Let $d_{\text{initial}} = \text{Run}\,(\text{Evaluate}_O(P_{\text{initial}}), \top, d_0)$ be the disk state reached by running the program P_{initial} against storage system O on the all-true (i.e., crash-free) crash schedule $\top$ starting from the empty disk d_0. A set of dependency rules R makes T crash consistent if, for all crash schedules $\mathbf{s}$ such that* $\text{Valid}_R(\mathbf{s}, P_{\text{main}})$ *is true,* $\text{Consistent}(\text{Run}\,(\text{Evaluate}_O(P_{\text{main}}), \mathbf{s}, d_{\text{initial}}))$ *holds.*

Example 3.1 Consider the `SingleEntry_TwoAppend` litmus test from Section 3.2. Interpreting the initial and main programs gives two traces:

$$\begin{aligned} \mathit{Interpret}(P_{initial}) = [&\texttt{write}(1, \texttt{to_block}((0, 42)), \langle \text{log}, 0 \rangle), \\ &\texttt{write}(0, \texttt{to_block}((1, 2)), \langle \text{superblock}, 0 \rangle)] \\ \mathit{Interpret}(P_{main}) = [&\texttt{write}(2, \texttt{to_block}((1, 81)), \langle \text{log}, 1 \rangle), \\ &\texttt{write}(0, \texttt{to_block}((1, 3)), \langle \text{superblock}, 1 \rangle), \\ &\texttt{write}(3, \texttt{to_block}((2, 37)), \langle \text{log}, 2 \rangle), \\ &\texttt{write}(0, \texttt{to_block}((1, 4)), \langle \text{superblock}, 2 \rangle)] \end{aligned}$$

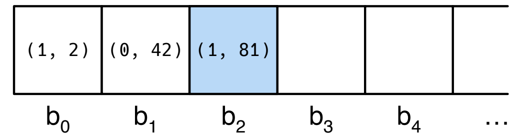
Let $\mathbf{s} = [s_1, s_2, s_3, s_4]$ be a crash schedule for P_{main}. Applying the two synthesized rules from Section 3.2 restricts the valid crash schedules (Definition 3.4):

- superblock $\leadsto_{=}$ log requires $s_2 \longrightarrow s_1$ and $s_4 \longrightarrow s_3$.
- superblock $\leadsto_{>}$ superblock requires $s_4 \longrightarrow s_2$.

Combined, these constraints yield seven valid crash schedules. Besides the two trivial crash schedules $\mathbf{s} = \top$ and $\mathbf{s} = \bot$, the other five crash schedules yield five distinct disk states:

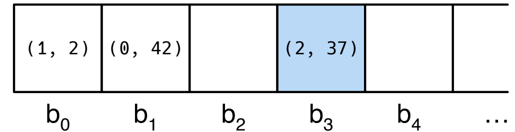
(1) $[s_1 = \top, s_2 = \bot, s_3 = \bot, s_4 = \bot]$

(only the first log block is on disk)



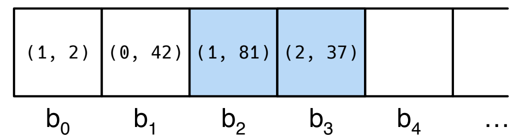
(2) $[s_1 = \bot, s_2 = \bot, s_3 = \top, s_4 = \bot]$

(only the second log block is on disk)



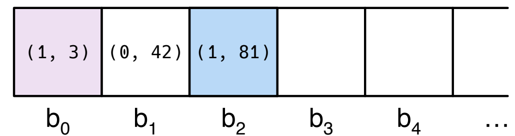
(3) $[s_1 = \top, s_2 = \bot, s_3 = \top, s_4 = \bot]$

(both log blocks are on disk)



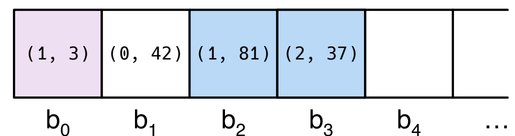
(4) $[s_1 = \top, s_2 = \top, s_3 = \bot, s_4 = \bot]$

(the first log block and first superblock write are on disk)



(5) $[s_1 = \top, s_2 = \top, s_3 = \top, s_4 = \bot]$

(the first log block, first superblock write, and second log block are on disk)

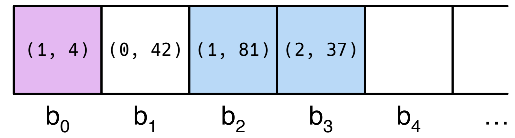


Each of these states satisfies the key-value store's crash-consistency predicate *Consistent*(d) defined in Section 3.2, as in each case the superblock's `head` and `tail` pointers refer only to log blocks that are also on disk. Some states result in data loss after the crash—for example, neither key can be retrieved from crash state (1) above, as the superblock is empty—but these states are still consistent (i.e., they satisfy the log's representation invariant). This set of two rules therefore makes the `SingleEntry_TwoAppend` litmus test crash consistent according to Definition 3.5.

If the second rule superblock $\rightsquigarrow_{>}$ superblock was excluded, the rule set with one remaining rule allows 2 additional crash states:

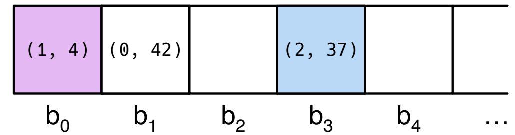
(6) $[s_1 = \top, s_2 = \bot, s_3 = \top, s_4 = \top]$

(the first log block, second log block, and second superblock write are on disk)



(7) $[s_1 = \bot, s_2 = \bot, s_3 = \top, s_4 = \top]$

(the second log block and second superblock write are on disk)



State (6) satisfies the crash-consistency predicate despite losing the first superblock write, as the second superblock write already contains the effects of the first one. However, state (7) violates the crash-consistency predicate: the first log block is invalid, but is included in the range between the superblock's `head` and `tail` pointers. The set containing only the first rule therefore does not make `SingleEntry_TwoAppend` crash consistent.

3.4 Dependency Rule Synthesis

This section describes the DepSynth synthesis algorithm, which automatically generates a set of dependency rules that are sufficient to guarantee crash consistency for a set of litmus tests. It formalizes the dependency rule synthesis problem, gives an overview of DepSynth's approach to synthesizing dependency rules, and then presents the core DepSynth algorithm (Fig. 3.2).

3.4.1 Problem Statement

DepSynth solves the problem of finding a *single* set of dependency rules R that makes every litmus test T in a set of tests $\mathcal{T}$ crash consistent (Definition 3.5). While Definition 3.5 suffices to find a set of rules R that guarantees crash consistency, it does not rule out *cyclic* solutions that

cannot be executed on real hardware. For example, consider a program P where $Evaluate_O(P) = [\mathtt{write}(a_1, v_1, \langle n_1, t_1\rangle), \mathtt{write}(a_2, v_2, \langle n_2, t_2\rangle)]$. The set of rules $R = \{n_1 \rightsquigarrow_= n_2, n_2 \rightsquigarrow_= n_1\}$ makes P crash consistent. These two rules do not admit any valid crash schedules other than the trivial $s = \top$ and $s = \bot$ schedules, as Definition 3.4 forces $s_1 = s_2$. In effect, crash consistency for P requires both writes to happen "at the same time". But on real disks the level of write atomicity is only a single data block, so there is no way for both writes to happen at the same time. To rule out cyclic solutions, we follow the example of happens-before graphs [Lam78] from distributed systems and memory consistency, and require the set of synthesized dependency rules R to be *acyclic.*

3.4.2 The DepSynth Algorithm

The DepSynth algorithm (Fig. 3.2) takes as input a storage system implementation $\mathcal{O}$, a set of litmus tests $\mathcal{T}$, and a crash-consistency predicate *Consistent.* Given these inputs, it synthesizes a set of dependency rules that is acyclic and sufficient to make all tests $\mathcal{T}$ crash consistent.

DepSynth does not try to generate a sufficient set of dependency rules for all tests in $\mathcal{T}$ at once, since this would require a prohibitively expensive search over large happens-before graphs. Instead, it works incrementally: at each iteration of its top-level loop, DepSynth chooses a single test T that is not made crash consistent by the current candidate set of dependency rules (line 4 in Fig. 3.2), invokes the procedure RulesForTest (Section 3.4.3) to synthesize dependency rules that make T crash consistent, and adds the new rules to the candidate set (line 13). Working incrementally reduces the number of litmus tests for which DepSynth needs to synthesize rules; for example, in Section 3.6.1 we show that only 10 of 16,250 tests were passed to RulesForTest to synthesize a sufficient set of dependency rules for a production key-value store. This reduction relieves developers from being selective about the set of litmus tests they supply to DepSynth, and makes it possible to, for example, use the output of a fuzzer or random test generator as input.

```
1  function DepSynth(O, 𝒯, Consistent)
2      R ← {}
3      loop
4          T ← NextTest(𝒯, R, O, Consistent)
5          if T = ⊥ then                          ▹ R makes all tests in 𝒯 crash consistent
6              return R
7          end if
8          𝒯 ← 𝒯 \ T
9          R′ ← RulesForTest(T, O, Consistent)
10         if R′ = ⊥ then
11             return UNSAT                        ▹ No rules can make T crash consistent
12         end if
13         R ← R ∪ R′
14         if ¬Acyclic(R) then                    ▹ Fail if new rules create a cycle in the rule set
15             return UNKNOWN
16         end if
17     end loop
18 end function

19 function NextTest(𝒯, R, O, Consistent)
20     for T ∈ 𝒯 do
21         if ¬CrashConsistent(T, R, O, Consistent) then
22             return T
23         end if
24     end for
25     return ⊥
26 end function

                                                   ▹ Check Def. 3.5 with an SMT solver
27 function CrashConsistent(T = ⟨P_initial, P_main⟩, R, O, Consistent)
28     d_initial ← Run (Evaluate_O(P_initial), ⊤, d_0)
29     return ∀s. Valid_R(s, P_main) ⇒ Consistent(Run (Evaluate_O(P_main), s, d_initial))
30 end function
```

Figure 3.2: The DepSynth algorithm takes as input a storage system implementation $\mathcal{O}$, a set of litmus tests $\mathcal{T}$, and a crash-consistency predicate *Consistent*, and returns an acyclic set of dependency rules that make all tests in $\mathcal{T}$ crash consistent (Definition 3.5). The search synthesizes dependency rules for one litmus test at a time. If the rules generated for two or more tests result in a cycle, this algorithm fails; Section 3.4.4 discusses an extension for continuing the search for an acyclic solution.

However, because the rules for each test are generated independently, it is possible for the union of the generated rules to contain a cycle—even if the rules for each individual test do not—and so be an invalid solution (Section 3.4.1). The algorithm in Fig. 3.2 returns UNKNOWN if such a cycle is found. We have not seen this failure mode occur for the storage systems we evaluated (Section 3.6), but it is possible in principle. In Section 3.4.4, we explain how to extend DEPSYNTH to recover from cycles by generalizing RULESFORTEST to synthesize rules for multiple tests at once.

DEPSYNTH delegates checking for crash consistency to the procedure CRASHCONSISTENT (line 27), which takes as input a single litmus test and a set of dependency rules, and checks whether the rules make the test crash consistent according to Definition 3.5. This procedure uses symbolic evaluation of the storage system implementation $\mathcal{O}$ to generate the logical encoding described in Section 3.3.3, and solves the resulting formulas using an off-the-shelf SMT solver [NPB15].

3.4.3 Synthesizing Dependency Rules with Happens-Before Graphs

The core of the DEPSYNTH algorithm is the RULESFORTEST procedure in Fig. 3.3, which takes as input a litmus test T, a storage system implementation $\mathcal{O}$, and a crash-consistency predicate *Consistent*, and synthesizes a set of dependency rules that makes T crash consistent. RULESFORTEST frames the rule synthesis problem as a search over *happens-before graphs* [Lam78] on the writes performed by the test. An edge (w_1, w_2) between two writes in a happens-before graph says that write w_1 must persist to disk before write w_2. Happens-before graphs and dependency rules have a natural correspondence: if a happens-before graph includes an edge (w_1, w_2), a dependency rule $n_2 \rightsquigarrow_p n_1$ that matches the writes' labels is sufficient to enforce the required ordering. RULESFORTEST searches for a minimal, acyclic happens-before graph that is sufficient to ensure crash consistency for T, and then syntactically generalizes that happens-before graph into

```
31 function RulesForTest(T = ⟨P_initial, P_main⟩, O, Consistent)
32     W ← {w | w ∈ Evaluate_O(P_main)}
33     return Phase1(𝒯, [], W, O, Consistent)
34 end function

35 function Phase1(T, order, W, O, Consistent)      ▷ Search for total orders over writes
36     if W = ∅ then
37         G ← {(order[i], order[j]) | 0 ≤ i < j < |order|}
38         return Phase2(𝒯, G, O, Consistent)       ▷ G is a total order; minimize it in Phase 2
39     end if
40     for w ∈ W do
41         order′ ← order + [w]
42         W′ ← W \ {w}
43         G ← {(order[i], order[j]) | 0 ≤ i < j < |order|} ∪
                 {(w1, w2) | w1 ∈ order ∧ w2 ∈ W} ∪
                 {(w1, w2) | w1, w2 ∈ W}
44         if ¬CrashConsistent(T, RulesForGraph(G), O, Consistent) then
45             continue
46         end if
47         R ← Phase1(T, order′, W′, O, Consistent)
48         if R ≠ ⊥ then
49             return R
50         end if
51     end for
52     return ⊥
53 end function
```

Figure 3.3: The algorithm for generating sufficient dependency rules for a litmus test T searches the space of happens-before graphs over the writes performed by T. The first phase searches for total orders over the writes that are sufficient for crash consistency. Once such a total order is found, the second phase (shown in Figure 3.4) removes edges from it until the happens-before graph is minimal.

a set of dependency rules.

RULESFORTEST searches for a happens-before graph by first finding a *total order* on the writes that makes T crash consistent (PHASE1), and then searching for a minimal *partial order* within this total order that is both sufficient for crash consistency and yields an acyclic set of dependency rules (PHASE2). The algorithm is exhaustive: it tries all total orders and all minimal partial orders within a total order, until it finds a solution or fails because a solution does not exist.

RULESFORTEST builds on the observation that crash consistency (Definition 3.5) is monotonic with respect to the subset relation on dependency rules—if a set of dependency rules R is not sufficient for crash consistency, then no subset of R is sufficient either:

Theorem 3.1 (Monotonicity of crash consistency) *Let T be a litmus test and R a set of dependency rules for a storage system $\mathcal{O}$. If R does not make T crash consistent (according to Definition 3.5), then no subset $R' \subset R$ can make T crash consistent.*

Proof 3.1 (Proof sketch) *If R does not make T crash consistent, there exists a valid crash schedule $\mathbf{s}$ (Definition 3.4) that does not satisfy the crash consistency predicate* Consistent. *By Definition 3.4, each rule in R only adds additional constraints on the possible valid crash schedules. Removing a rule from R therefore only allows more valid crash schedules, and so if $\mathbf{s}$ was a valid crash schedule for R, it is also a valid crash schedule for any subset of R.*

RULESFORTEST applies this property by checking crash consistency for a happens-before graph G before exploring any subgraphs of G; if G is not sufficient, then neither is any subgraph of G, and so that branch of the search can be skipped.

Total order search. PHASE1 (line 35) explores all possible total orders over the writes in T that are sufficient for crash consistency. At each recursive call, the list *order* represents a total order over some of T's writes, and the set W contains all writes not yet added to that order. PHASE1 tries

```
54 function Phase2(T, G, O, Consistent)        ▷ Minimize graph G by removing individual edges
55     R ← RulesForGraph(G)
56     if ¬CrashConsistent(T, R, O, Consistent) then
57         return ⊥
58     end if
59     for (w1, w2) ∈ G do                                   ▷ Try removing each edge from G
60         G′ ← G \ {(w1, w2)}
61         R′ ← Phase2(T, G′, O, Consistent)
62         if R′ ≠ ⊥ then
63             return R’
64         end if
65     end for
66     if Acyclic(R) then
67         return R                   ▷ G makes T crash consistent and no subgraph of G suffices
68     else
69         return ⊥
70     end if
71 end function
```

Figure 3.4: The second phase of the dependency rules generation algorithm. This phase greedily removes edges to from the happens-before graph to arrive at a minimal graph.

to add each write in W to the end of the total order. Each time, it checks whether the new total order leads to a crash consistency violation (line 44) and if so, prunes this branch of the search. For Phase1 to be complete, this check must behave angellically for the writes in W that have not yet been added to the order—if there is *any possible* set of dependency rules for the remaining writes that would succeed, the check must succeed. We make the check angelic by including every possible dependency rule for the remaining writes (line 43). If the test cannot be made crash consistent even with every possible rule included, then by Theorem 3.1 no subset of those rules (i.e., formed by completing the rest of the total order) can succeed either, so the prefix is safe to prune. Phase1 continues until every write has been added to the total order and then moves to Phase2 to further reduce the happens-before graph.

```
72 function RULESFORGRAPH(G)        ▷ Generalize a happens-before graph into dependency rules
73     R ← {}
74     for (w_1, w_2) ∈ G do
75         ⟨n_1, t_1⟩ ← LABEL(w_1)      ▷ Get label l_1 = ⟨n_1, t_1⟩ for write w_1 = write(a_1, s_1, l_1)
76         ⟨n_2, t_2⟩ ← LABEL(w_2)
77         if t_1 < t_2 then
78             R ← R ∪ {n_2 ⇝_> n_1} ▷ Invert the order, as a rule n_a ⇝_p n_b says n_a happens after n_b
79         else if t_1 = t_2 then
80             R ← R ∪ {n_2 ⇝_= n_1}
81         else
82             R ← R ∪ {n_2 ⇝_< n_1}
83         end if
84     end for
85     return R
86 end function
```

Figure 3.5: The procedure for generalizing a happens-before graph G into dependency rules constructs (at most) one rule for each edge.

Partial order search. Starting from a happens-before graph G that reflects a total order over all writes in T, PHASE2 (Figure 3.4) removes edges from the graph until it is minimal, i.e., removing any further edges would violate crash consistency. PHASE2 removes one edge at a time from the graph G (line 60), checks if the graph remains sufficient for crash consistency (line 56), and if so, recurses to remove more edges. By greedily removing one edge at a time, PHASE2 is guaranteed to find a minimal result, and because PHASE2 considers removing every possible edge from G (except those that cannot lead by solutions by Theorem 3.1), it is complete—if an acyclic solution exists, PHASE2 will reach it.

Generating rules from happens-before graphs. The RULESFORTEST search operates on happens-before graphs, but its goal is to synthesize dependency rules (Definition 3.1). The RULES-FORGRAPH procedure (Fig. 3.5) bridges this gap by taking as input a happens-before graph G and returning a set of dependency rules R that are sufficient to enforce the ordering requirements that

G dictates. RulesForGraph uses a simple syntactic approach to generate a rule for each edge in G: if $(w_1, w_2) \in G$, where writes w_1 and w_2 have labels $l_1 = \langle n_1, t_1 \rangle$ and $l_2 = \langle n_2, t_2 \rangle$, respectively, then it generates a rule of the form $n_2 \rightsquigarrow n_1$ (reversing the order because G is a happens-before graph but dependency rules are happens-*after* edges). To choose an epoch predicate for the generated rule, we compare the two epochs t_1 and t_2 and select the predicate that would make the rule match the labels l_1 and l_2.

This approach can lead to rules that are too general, as some rules it generates may only need to apply to certain individual epochs but will instead apply to all epochs that match the predicate. Overly general rules risk sacrificing performance by preventing reordering or caching optimizations that would be safe. However, this same generality also allows RulesForTests to avoid overfitting to the input litmus tests. In Section 3.6.1 we show that generated rules generalize well in practice (i.e., are not overfit), and that they filter out few additional schedules compared to expert-written rules.

Properties of RulesForTest. The RulesForTest algorithm is *sound*: all paths that return a solution are guarded by checks of crash consistency and of acyclicity, and so satisfy the requirements of Section 3.4.1. RulesForTest is also *complete*: each of Phase1 and Phase2 are complete, as discussed above, and so together form a complete search over the space of total orders. Every possible acyclic solution must be a subgraph of some total order, since the transitive closure of edges in any happens-before graph is a (strict) partial order, and so exploring all total orders suffices to reach any possible acyclic solution. Finally, RulesForTest is *minimal*, in the sense that removing any rule from a returned set R would violate crash consistency. Phase2 continues removing edges from a candidate graph G until Theorem 3.1 says it cannot be made smaller, and is therefore guaranteed to find a minimal happens-before graph. Every rule in R is justified by (at least) one edge in that graph, and since dependency rules cannot overlap (in Definition 3.1, the

possible epoch predicates are disjoint), removing any rule would incorrectly allow reordering of its corresponding edge(s).

Example 3.2 Consider running RULESFORTEST for the simple log-structured key-value store and `SingleEntry_TwoAppend` litmus test from Section 3.2. From Example 3.1 we know that this test produces a set W of four writes:

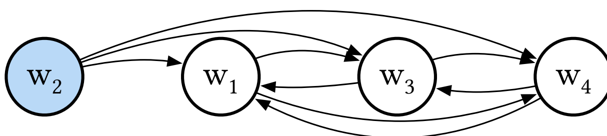
$$w_1 = \texttt{write}(2, \texttt{to_block}((1, 81)), \langle \text{log}, 1 \rangle),$$

$$w_2 = \texttt{write}(0, \texttt{to_block}((1, 3)), \langle \text{superblock}, 1 \rangle),$$

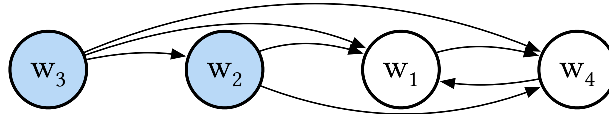
$$w_3 = \texttt{write}(3, \texttt{to_block}((2, 37)), \langle \text{log}, 2 \rangle),$$

$$w_4 = \texttt{write}(0, \texttt{to_block}((1, 4)), \langle \text{superblock}, 2 \rangle)$$

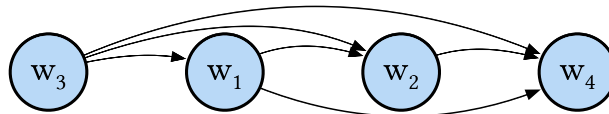
PHASE1 first chooses the first write to add to the total order. Suppose it chooses w_2. This choice results in the following graph G at line 43 (shaded nodes are in *order*; white nodes are in W):



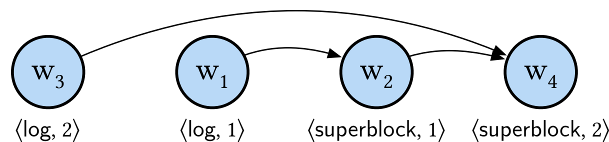
The check at line 44 finds that this graph is not crash consistent: it allows a crash schedule where w_2 is on disk but no other writes are, which violates the crash-consistency predicate as w_2 is a superblock write pointing to a log block that is not on disk. PHASE1 therefore continues (line 45), which prunes any total order that starts with $[w_2]$ from the search, and chooses a next write to consider, say w_3. The total order starting with $[w_3]$ does pass the crash consistency check, so PHASE1 recurses with *order* $= [w_3]$ and $W = \{w_1, w_2, w_4\}$. In this recursive call, suppose we again first choose w_2 to add to the total order. This choice results in the following graph G:



Again, line 44 finds that this graph is not crash consistent, for the same reason as before (superblock write w_2 can be on disk when log write w_1 is not), and so the search continues, pruning any total order that starts with $[w_3, w_2]$. Suppose it next chooses w_1 to add to the total order. This choice succeeds, making the recursive call with *order* $= [w_3, w_1]$ and $W = \{w_2, w_4\}$. From here, any choice PHASE1 makes will succeed. Supposing it choses w_2 first, PHASE1 eventually reaches line 38 and continues to PHASE2 with the following initial graph G:



PHASE2 proceeds by trying to remove one edge at a time from G. Suppose it first chooses to remove edge (w_3, w_1), and so recurses at line 61 on the graph $G' = G \setminus \{(w_3, w_1)\}$. This graph still ensures crash consistency at line 56, as writing w_1 before w_3 does not affect consistency. The recursion can continue twice more by choosing and successfully removing edges (w_3, w_2) and then (w_1, w_4) as well, eventually reaching line 59 with the following graph G (now with write labels shown):



From here, the loop in PHASE2 now tries to remove each of the three remaining edges, but each attempted G' violates crash consistency and so returns $\perp$ from the next recursive call. PHASE2 therefore exits the loop with the above graph G, which we now know is minimal as no further

edges can be removed. Applying RULESFORGRAPH to G yields the two rules from Section 3.2:

$$\text{superblock} \rightsquigarrow_{=} \text{log} \qquad \text{from edges } (w_3, w_4) \text{ and } (w_1, w_2)$$

$$\text{superblock} \rightsquigarrow_{>} \text{superblock} \qquad \text{from edge } (w_2, w_4).$$

3.4.4 Resolving Cycles in Dependency Rules

The top-level DEPSYNTH algorithm generates rules for each litmus test independently. Even though the rules generated for each test are guaranteed to be acyclic, it is possible for the *union* of those rules to contain a cycle, and so violate the requirements of Section 3.4.1. In practice, we have not seen this happen for the storage systems we evaluate in Section 3.6, and so the version of DEPSYNTH presented in Fig. 3.2 fails if the synthesized rules contain a cycle.

To handle cyclic rules, RULESFORTEST can be extended to support synthesizing rules for multiple litmus tests at once. This extension adds the writes from *all* the tests into the set of writes W, searches for a total order over that entire set in PHASE1, and then searches for a minimal happens-before graph over the entire set in PHASE2. Edges between writes from different tests cannot influence the crash consistency of individual tests (in Definition 3.4 they will just lead to spurious additional implications), and they will eventually be removed by PHASE2, creating a forest of disjoint happens-before graphs. PHASE2 is therefore guaranteed to return an acyclic set of dependency rules for all the tests it was provided.

In the limit, DEPSYNTH could just invoke RULESFORTEST with its entire input set $\mathcal{T}$, but this would be prohibitively expensive for any non-trivial set of tests. Instead, our implementation resolves cycles in DEPSYNTH by identifying which individual litmus tests caused the cycle (i.e., which tests the rules in the cycle were generated from), and passes only that subset of tests to the extended RULESFORTEST.

3.5 Optimizations and Design Choices

We implement both the DepSynth algorithm and the storage systems we study in Section 3.6 in Rosette [TB14], an extension of Racket [FFF+18] with support for verification and synthesis. Using Rosette as our host language gives us symbolic evaluation of the storage system implementation for free, and simplifies implementing the CrashConsistent query in Fig. 3.2. The choice of Rosette and Racket is not fundamental; recent work has shown how to extend the symbolic evaluation approach to languages such as Python [SBTW16] or C [NBG+19] in which storage systems are more commonly implemented.

Ordering. The DepSynth algorithm in Fig. 3.2 is sensitive to the order in which NextTest chooses tests to generate dependency rules for. Our implementation chooses tests in increasing order of size, minimizing the number of happens-before graphs for RulesForTest to explore. Similarly, RulesForTest is sensitive to the order it considers writes (Phase1) and edges (Phase2). In both cases, we exploit the following observation: while an execution that persists writes in program order is not *required* to be crash consistent (e.g., because storage systems might selectively buffer or coalesce writes), it is often so in practice. RulesForTest therefore prefers to choose writes in Phase1 in program order, and prefers to remove edges in Phase2 that contradict program order.

Reducing solver queries. Both Phase1 and Phase2 in Fig. 3.3 have symmetry in their search space: for a fixed pair of writes w_1 and w_2, there are many different branches of Phase1 that try to order w_2 after w_1, and many different branches of Phase2 that try to remove the edge (w_1, w_2) from a happens-before graph. If we can determine ahead of time that such a choice for those writes is always doomed to fail, we can avoid considering these choices at all and so save the cost of an SMT solver query by CrashConsistent. Our implementation of RulesForTest uses

an SMT solver to pre-compute a set of *necessary* ordering edges—edges which *must* be in the happens-before graph—and uses that set to short-circuit CRASHCONSISTENT.

3.5.1 Attempting graph search with a $\mathcal{T}_{ord}$ solver.

One bottleneck in DEPSYNTH comes from the time it takes to search for a minimal happens-before graph for a given litmus test. This section describes a strategy that we explored to limit this bottleneck. Specifically, we encode the happens-before graph search problem in the theory of ordering consistency ($\mathcal{T}_{ord}$) [HSF21] and execute the search with an existing $\mathcal{T}_{ord}$ solver. Ultimately, we found that this different search strategy did not improve the overall runtime, but we find this negative result worth reporting.

As described in Section 3.4.3, the RULESFORTEST algorithm finds dependency rules for a litmus test by searching for a total order over writes, reducing the total order to a minimal happens-before graph, then generalizing that graph into rules. The reduction to graph and generalization into rules steps are both straightforward, but since the space of possible total orders grows exponentially with the number of writes, finding valid orders over writes can be exceedingly slow for large tests. While the pruning techniques presenting in Section 3.4 and the heuristics over prioritizing program order help mitigate the runtime cost of this search, there remains space for further improvement.

In an attempt to speed up the overall search time for total orders over writes, we encode the problem in $\mathcal{T}_{ord}$ and offload to a $\mathcal{T}_{ord}$ solver. $\mathcal{T}_{ord}$ defines predicates for describing partial-ordering constraints on reads and writes for multi-threaded programs, but can also be used to describe orderings of storage system writes. Specifically, $\mathcal{T}_{ord}$ introduces the predicate $\prec$, where $w_i \prec w_j$ encodes the fact that write w_i occured before write w_j. To encode our search in $\mathcal{T}_{ord}$, we describe the existence of each happens-before graph edge between write events e_i and e_j as the boolean

variable d_{ij}, then introduce the following constraint: $d_{ij} \rightarrow e_i \prec e_j$. By solving these constraints along with the user-defined constraints specifying crash consistency for the storage system, we arrive at a total order over all writes for the given litmus test.

Existing work defines a DPLL(T)-style solver for $\mathcal{T}_{ord}$ [HSF21], which we use for solving the constraints specified above and determining a total order over writes for single litmus tests. However, by doing so, we empirically found that the solver slightly diminished the performance of DepSynth for all three of our evaluated case studies (in the case of ShardStore, slowing the rules search from 49 minutes to 55 minutes). Intuitively, we believe this performance degredation is an artifact of bad initial assignments from the SAT solver and a resulting a large number of iterations in the DPLL(T) loop. For storage systems with increasingly complex behavior, DepSynth may require much larger litmus tests, which may make the benefits of searching for orders with $\mathcal{T}_{ord}$ more apparent. However, for the case studies considered in this chapter, we found our original search algorithm from Section 3.4.3 to outperform the $\mathcal{T}_{ord}$-directed search.

3.5.2 Faster Consistency Checks for Litmus Tests

As Section 3.6.1 discusses, a large majority of the litmus tests seen during the DepSynth algorithm do not result in new dependency rules (DepSynth generated new rules for only 10 of 16,250 litmus tests seen for ShardStore). However, for each litmus test in the input set, DepSynth must perform a check that the accumulated dependency rules guarantee the crash consistency of the test. For synthesizing rules in ShardStore, these take up 45 minutes (93%) of the overall search time.

Decomposing crash consistency queries. As Section 3.4.2 explains, DepSynth offloads a query to an SMT solver when it first encounters a new litmus test to determine whether or not the current set of dependency rules satisfy the crash consistency property for that test. In practice, this query is encoded by representing the *committed status* of each write from a litmus test as a boolean

variable and each edge in the happens-before graph as an implication between those variables. For tests that generate a large happens-before graph with respect to the current dependency rules, the query can take tens of seconds.

Experimentally, we observed that for these queries with long runtimes, decomposing the query by concretizing one or more write-committed variables drastically speeds up solving time. (averaging 2× and at most 56× speedup for a single litmus test). Specifically, considering the set of write-committed variables V, the happens-before graph $G \subset V \times V$, and the original encoding for the litmus test crash consistency check $P(G)$, we choose a variable $v_0 \in V$ and issue the following two queries:

1. $v_0 \land \forall v \in V.(v, v_0) \in G \rightarrow v) \land P(G)$

2. $\neg v_0 \land \forall v \in V.(v, v_0) \in G \rightarrow \neg v) \land P(G)$

The litmus test is crash consistent with respect to the dependency rules if and only if both queries succeed. This concretization can be repeated for any number of variables, resulting in 2^n queries for n concretized variables. Ultimately, this optimization gives a speedup of 1.4× for the ShardStore case study discussed in Section 3.6.1, reducing end-to-end runtime from 49 minutes to 34 minutes.

Choosing Variables to Concretize. Intuitively, this decomposition results in faster solving times for two reasons. First, this optimization can be seen as a pre-processing variable selection step. By taking advantage of the relationship between nodes in the happens-before graph (which is abstracted away in the constraints sent to the solver), we are effectively telling the solver to select a good first variable for branching. Second, concretizing write-committed variables can also indirectly simplify the crash consistency predicate for the litmus test. The crash consistency predicate is a function over disk states, and the disk state itself depends on the write-committed variables. Thus, by assigning concrete values to some of those variables, constraints representing

the crash consistency predicate over the disk state may be simplified or eliminated.

With this intuition in mind, we chose concretization variables that will most effectively simplify the constraints given to the solver. We found that for a litmus test with N writes, choosing $log(N)$ variables to concretize gave the best results. Moreover, the choice of which variables to concretize is determined by the happens-before graph over the litmus test writes (w.r.t. DEPSYNTH's current candidate rules set). Specifically, we choose the $log(N)$ variables whose corresponding node in the happens-before graph has the most ancestors and descendants.

3.6 Evaluation

This section answers three questions to demonstrate the effectiveness of DEPSYNTH:

1. Can storage system developers use DEPSYNTH to synthesize dependency rules for a realistic storage system rather than implementing their own crash-consistency approach by hand? (§3.6.1)

2. Can DEPSYNTH help storage system developers avoid crash-consistency bugs? (§3.6.2)

3. Does DEPSYNTH's approach support a variety of storage system designs? (§3.6.3)

3.6.1 ShardStore Case Study

To show that developers can use DEPSYNTH to build realistic storage systems, we implemented a key-value store that follows the design of ShardStore [BJA+21], the exabyte-scale production storage node for the Amazon S3 object storage service.

Implementation. The first step in using DEPSYNTH is to implement the storage system itself. ShardStore's on-disk representation is a log-structured merge tree (LSM tree) [OCGO96], but

with values stored outside the tree in a collection of extents. Our ShardStore-like storage system implementation consists of 1,200 lines of Racket code, including five operations: the usual `put`, `get`, and `delete` operations on single keys, as well as a garbage collection `clean` operation that evacuates all live objects in one extent to another extent, and a `flush` operation that persists the LSM tree memtable to disk. Our implementation does not handle boundary conditions such as running out of disk space or objects too large to fit in one extent, but is otherwise faithful to the published ShardStore design. As a crash consistency predicate, we wrote a checker that validates all expected objects are accessible by `get` after a crash, and that the on-disk LSM tree contains only valid pointers to objects in extents.

Synthesis. With a storage system implementation in hand, a developer can use DEPSYNTH to synthesize dependency rules that make the system crash consistent. DEPSYNTH takes as input a set of litmus tests—we randomly generated 16,250 litmus tests for the ShardStore-like system, ranging in length from 1 to 16 operations. Executing these tests against the system led to an average of 7.2 and a maximum of 20 disk writes per test. Given these inputs, DEPSYNTH synthesized a set of 20 dependency rules for ShardStore in 49 minutes. To find a correct solution for all 16,250 litmus tests, the DEPSYNTH algorithm invoked the RULESFORTEST procedure (line 9 in Fig. 3.2) only 10 times, showing that DEPSYNTH's incremental approach is effective at reducing the search space.

Comparison to an existing implementation. ShardStore is an existing production system and already supports crash consistency. Its implementation does not use a dependency-rule language like in DEPSYNTH. Instead, it implements a soft-updates approach [GP94] by constructing dependency graphs (i.e, happens-before graphs) at run time and sequencing writes to disk based on those graphs, similar to patchgroups in Featherstitch [FMK+07]. We therefore compare our synthesized rules against ShardStore's dependency graphs to see how well DEPSYNTH may replace

Table 3.1: Valid schedules allowed by the production ShardStore service versus the dependency rules we synthesized for our ShardStore-like reimplementation. A schedule allowed only by one implementation means either that implementation is not crash consistent (it allows a schedule it should forbid) or it admits more reordering opportunities (it allows a schedule it should allow). "Fixed" results are after fixing two issues in ShardStore (one consistency, one performance) that we identified by manually inspecting the "original" schedules.

			Allowed by both		Allowed only by DEPSYNTH		Allowed only by ShardStore	
Test	Test Length	Writes	Original	Fixed	Original	Fixed	Original	Fixed
T_1	1	2	3	3	0	0	0	0
T_2	2	6	7	14	7	0	3	3
T_3	5	1	2	2	0	0	0	0
T_4	5	7	8	15	7	0	3	3
T_5	4	7	11	29	9	0	9	0
T_6	5	5	6	12	2	0	4	0
T_7	7	5	5	11	2	0	5	1
T_8	10	5	6	12	2	0	4	0
T_9	16	6	8	22	2	0	12	0
T_{10}	13	9	21	41	20	0	9	9

an expert-written crash consistency implementation.

For each of the 10 tests that DEPSYNTH used while synthesizing dependency rules for ShardStore, we used an SMT solver to compute the set of valid crash schedules (Definition 3.4) according to those synthesized dependency rules. We then executed the same test using the production ShardStore implementation, collected the run-time dependency graph it generated, and used an SMT solver to compute the set of valid crash schedules according to that graph. Given these two sets of crash schedules, we computed the set intersection and difference to classify them into three groups: schedules allowed by both implementations (i.e., both implementations agree), and schedules allowed only by one or the other implementation (i.e., the two implementations disagree).

Table 3.1 shows the results of this classification across the 10 litmus tests. Overall, the two implementations agree on the validity of an average of 87% of crash schedules. The remaining crash schedules are in two categories:

1. Schedules allowed only by DEPSYNTH mean either DEPSYNTH's rules allow some schedules that

are not crash consistent (a correctness issue in the synthesized rules) or ShardStore precludes some schedules that are crash consistent (a performance issue in ShardStore). We found that every schedule allowed by DepSynth is crash consistent, and that ShardStore inserts unnecessary edges in its dependency graphs, ruling out some reorderings that would be safe. These edges are not necessary to guarantee crash consistency of the overall storage system, and so DepSynth is correct to allow them. However, ShardStore engineers intentionally include these edges as they make the representation invariant for an on-disk data structure simpler, even though a more complex invariant that did not require these edges would still be sufficient for consistency. In other words, ShardStore engineers favored a stronger, simpler invariant in these cases, where DepSynth is able to identify opportunities for performance improvements.

2. Schedules allowed only by ShardStore mean either DepSynth's rules preclude some schedules that are crash consistent (meaning DepSynth's output is not optimal) or ShardStore allows some schedules that are not crash consistent (a correctness issue in ShardStore). 67% of these schedules are incorrectly allowed by ShardStore due to a rare crash-consistency issue that was independently discovered concurrently with this work. We have confirmed with ShardStore engineers that the issue was an unlikely edge case that could not lead to data loss, but could lead to "ghost" objects—resurrected pointers to deleted objects, where the object data has been (correctly) deleted, but the pointer still exists—which result in an inconsistent state. After fixing this issue in ShardStore, we manually inspected the remaining schedules it allowed and confirmed they are all cases where DepSynth's rules generate extraneous edges (i.e., the synthesized rules are not optimal), and the crash-consistency predicate we wrote for our ShardStore reimplementation agrees that all the resulting states are consistent.

After fixing the two ShardStore issues discussed above, the synthesized dependency rules agree with ShardStore on the validity of an average of 99% of crash schedules. The few remaining

schedules are ones that DepSynth's synthesized dependency rules conservatively forbid due to the coarse granularity of the dependency rule language. Overall, this study shows that DepSynth achieves similar results to an expert-written crash consistency implementation, and can help identify correctness and performance issues in existing storage systems.

Generalization. One risk for example-guided synthesis techniques like DepSynth is that they can overfit to the examples (litmus tests) and not actually ensure crash consistency on unseen test cases. DepSynth's design reduces this risk by using a simple dependency rule language (Definition 3.1) that cannot identify individual write operations. To test generalization, we randomly generated an additional 136,000 litmus tests for our ShardStore-like system. We also allowed these tests to be significantly longer than those used during synthesis—up to a maximum of 40 writes rather than the 20 in the input set of litmus tests. For each new test, we used the synthesized dependency rules to compute all valid crash schedules for the test, and found that every crash schedule resulted in a consistent disk state according to our crash consistency predicate. In other words, by limiting the expressivity of our dependency rule language, the rules we synthesize can generalize well beyond the tests they were generated from.

3.6.2 Crash-Consistency Bugs

To understand how effective DepSynth can be in preventing crash-consistency bugs, we surveyed all bugs reported by two recent papers [BJA+21, MMP+18] in three production storage systems for which a known fix is available. We manually analyze each bug and determine whether DepSynth could discover and prevent them.

Table 3.2 shows the results of our survey. In six cases, DepSynth could have prevented the bug by synthesizing a dependency rule to preclude a problematic reordering optimization. Each of these bugs had small triggering test cases, suggesting they would be reachable by a litmus-test-

Table 3.2: Sample crash-consistency bugs in three storage systems reported by two recent papers [BJA+21, MMP+18]. Each bug includes its identifier (bug number for ShardStore, kernel Git commit for btrfs and f2fs). Most of these bugs could have been prevented by using DEPSYNTH to automatically identify missing ordering requirements, but some crash-consistency issues are either not ordering related or are unlikely to be detected by DEPSYNTH's litmus-test-driven approach.

Storage system	Crash-consistency bug	Preventable by DEPSYNTH?
ShardStore	Inconsistency in extent allocation (#6)	Yes
ShardStore	Mismatch between soft and hard write pointers (#7)	Yes
ShardStore	Index entries persisted before target data (#8)	Yes
ShardStore	Crash consistency predicate too strong (#9)	No—specification bug
ShardStore	Data loss after UUID collision (#10)	No—unlikely to detect
btrfs	Extents deallocated too early in recovery (`bf50411`)	Yes
btrfs	Inode rename commits out of order (`d4682ba`)	Yes
f2fs	`fsync` failed after directory rename (`ade990f`)	Yes
f2fs	Wrong file size when zeroing file beyond EOF (`17cd07a`)	No—not reordering

based approach like ours. In the other three cases, our analysis shows that DEPSYNTH would not prevent the bug. One bug in ShardStore was a specification bug in which the crash consistency predicate was too strong. DEPSYNTH assumes that the crash consistency predicate is correct, and will miss specification bugs. Another bug in ShardStore involved a collision between two randomly generated UUIDs. While such a bug would be possible to find in principle using litmus tests, it would be very unlikely, and without a test that triggers the issue DEPSYNTH cannot preclude it. One bug in f2fs involved an incorrect file size being computed when zero-filling a file beyond its existing endpoint. This bug was a logic issue rather than a reordering one (i.e., occurring even without a crash), and so no dependency rule would suffice to prevent it. Overall, our analysis indicates that DEPSYNTH can prevent a range of ordering-related crash-consistency bugs, but other bugs would require a different approach.

3.6.3 Other Case Studies

In addition to the ShardStore case study, we have applied DepSynth to two other storage systems. The first is a modification of ShardStore with a different underlying disk hardware model. The second is a simple log-strucutred file system. DepSynth was effectively able to synthesize rules for both of these additional cases, demonstrating the generality of our approach.

ShardStore with SMR. Part of DepSynth's requirement for input storage system implementations is a model of how the underlying disk hardware behaves. For example, some storage devices automatically enforce an order over writes or may disallow particular orders of writes to the same extent on disk. In the case study of ShardStore discussed above, we use a disk model that represents a conventional magnetic-recording (CMR) hard disk. For this second application of DepSynth, we instead use a model of shingled magnetic recording (SMR) disks and synthesize rules for ShardStore under this new model.

To understand how this difference in disk models will affect the synthesized rules, we first give a brief overview of SMR disks. Like traditional CMR disks, SMR drives store data on platters that contain tracks read by a magnetic needle. Where the technologies differ is in the layout of the tracks: tracks in CMR disks are non-overlapping and lay parallel to each other, while tracks in SMR lay partially over one another. This allows SMR disks to store data more densely than their CMR counterparts, but requires that all writes to a single SMR extent be *appends*. In order to write data at a location before the append pointer in an SMR extend, the whole extent must first be wiped.

We ran DepSynth to synthesize rules for the ShardStore-like system with an underlying SMR disk model. To do so, we generated a new set of 16,250 litmus tests, each with a maximum of 16 operations. For this new system, DepSynth generated 18 in 37 minutes minutes. During this process, DepSynth invoked the RulesForTest procedure for 12 of the total 16,250 tests. These

results demonstrate that DEPSYNTH is resiliant to changes in hardware used by storage systems.

Log-Structured File System. We have used DEPSYNTH to implement a log-structured file system [RO91]. The file system supports five standard POSIX operations: `open`, `creat`, `write`, `close`, and `mkdir`. While our implementation is simple (300 lines of Racket code) compared to production file systems, it has metadata structures for files and directories, and so has its own subtle crash consistency requirements. For example, updates to data and inode blocks must reach the disk before the pointer to the tail of the log is updated. To synthesize dependency rules for this file system, we randomly generated 235 litmus tests with at most 6 operations. DEPSYNTH synthesized a set of 18 dependency rules in 12 minutes to make the file system crash consistent, and during the search, invokes RULESFORTEST for only 13 tests. This result shows that DEPSYNTH can automate crash consistency for storage systems other than key-value stores.

3.7 Related Work

Verified storage systems. Inspired by successes in other systems verification problems [Ler09, KEH+09], recent work has brought the power of automated and interactive verification to bear on storage systems as well. One of the main challenges in verifying storage systems is crash consistency, as it combines concurrency-like nondeterminism with persistent state. Yggdrasil [SBTW16] is a verified file system whose correctness theorem is a *crash refinement*—a simulation between a crash-free specification and the nondeterministic, crashing implementation. This formalization allows clients of Yggdrasil to program against a strong specification free from crashes, similar to our angelic crash consistency model. FSCQ [CZC+15] is a verified crash-safe file system with specifications stated in *crash Hoare logic*, which explicitly states the recovery behavior of the system after a crash. DFSCQ [CCK+17] extends FSCQ and its verification with support for crash-

consistency optimizations such as log-bypass writes and the metadata-only `fdatasync` system call. The DepSynth programming model separates crash consistency of these optimizations from the storage system itself, and so can simplify their implementation.

Another approach to verified storage systems is at the language level. Cogent [AHC+16] is a language for building storage systems with a strong type system that precludes some common systems bugs. A language-level approach like Cogent is complementary to DepSynth: Cogent provides a high-level language for implementing storage systems, while DepSynth provides a synthesizer for making those implementations crash consistent.

Crash-consistency bug-finding tools. Ferrite [BKL+16] is a framework for specifying *crash-consistency models*, which formally define the behavior of a storage system across crashes, and for automatically finding violations of such models in a storage system implementation. One way to specify these models is with litmus tests that demonstrate unintuitive behaviors; DepSynth builds on this approach by automatically synthesizing rules from such litmus tests. DepSynth also takes inspiration from Ferrite's synthesis tool for inserting `fsync` calls into litmus tests to make them crash consistent, but instead focuses on making the *storage system itself* crash consistent rather than the user code running on top of it. CrashMonkey [MMP+18] is a tool for finding crash-consistency bugs in Linux file systems. Chipmunk [LPE+23] extends the CrashMonkey approach to persistent-memory file systems by exploring finer-grained crash states to account for the byte-addressable nature of non-volatile memory. CrashMonkey exhaustively enumerates all litmus tests with a given set of system calls, runs them against the target file system, and then tests each possible crash state for consistency. Connecting CrashMonkey-like litmus test generation with DepSynth could provide developers with a comprehensive set of litmus tests for their system for free, lowering the burden of applying DepSynth. To give stronger coverage guarantees that do not depend on enumerating litmus tests, FiSC [YTEM04] and eXplode [YTEM06] use model

checking to find bugs in storage systems.

One advantage of bug-finding tools is that they are significantly easier to apply to production systems than heavyweight verification tools. Bornholt et al. [BJA+21] describe the use of lightweight formal methods to validate the crash consistency (and other properties) of ShardStore, the Amazon S3 storage node that we study in Section 3.6.1. Their approach applies property-based testing to automatically find and minimize litmus tests that demonstrate crash-consistency issues. DepSynth takes this idea one step further by automatically *fixing* such issues once they are found.

Program synthesis for systems code. Transit [URD+13] is a tool for automatically inferring distributed protocols such as those used for cache coherence. It guides the search using *concolic* snippets [SMA05]—effectively litmus tests that can be partially symbolic—and finds a protocol that satisfies those snippets for *any* ordering of messages. MemSynth [BT17] is a program synthesis tool for automatically constructing specifications of memory consistency models. MemSynth takes similar inputs to DepSynth—a set of litmus tests and a target language—and its synthesizer generates and checks happens-before graphs for those tests. Adopting MemSynth's aggressive inference of partial interpretations [TJ07] to shrink the search space of happens-before graphs would be promising future work.

3.8 Conclusion

DepSynth offers a new programming model for building crash-consistent storage systems. By offering a high-level angelic programming model for crash consistency, and automatically synthesizing low-level dependency rules to realize that model, DepSynth lowers the burden of building reliable storage systems. We believe that this work presents a promising direction for building systems software with the aid of automatic programming tools to resolve challenging nondeterminism and persistence problems.

Chapter 4

Conclusion

This dissertation has presented two tools — JitSynth and DepSynth — along with new metaprogramming abstractions and system decompositions that enable the tools to automate system design and implementation for their respective domains. Chapter 2 demonstrated how JitSynth frames the synthesis task as a search for per-instruction compilers over *abstract register machines*, taking advantage of new minicompiler metasketches to efficiently synthesize real-world JITs. In Chapter 3, we introduced a new *angelic* programming model for crash consistency, along with a tool DepSynth that synthesizes *dependency rules* to enforce crash consistency under this model. We also evaluated DepSynth on ShardStore, a production key-value storage system at Amazon, to show the effectiveness of this technique. Both works demonstrate the effectiveness of program synthesis in designing correct and performant systems.

There are several interesting directions for extending the work presented in this thesis. One is expanding on JitSynth to allow for some class of compiler optimizations. In order to further improve the runtime performance of synthesized compilers, JitSynth could use superoptimization techniques to minimize the size of discovered minicompilers. Exploring how to best apply these techniques to improve JitSynth's effectiveness is a promising direction for future work. Another

interesting direction lies in further improving DEPSYNTH's usability. Existing techniques for litmus test generation and partial interpretation of litmus tests [MMP+18, BT17] could allow DEPSYNTH to synthesize dependency rules more quickly. Finally, my overall thesis would be further supported by a larger body of synthesis tools for a wide array of modern systems. Chapter 3 discusses existing works in synthesis for techniques for both memory models and distributed protocols, but other promising system domains for program synthesis include blockchain staking protocols and security monitors for IFC kernels. Security and liveness properties for staking protocols are specified in a fairly straightforward manner, but complex invariants on the blockchain state make implementing staking mechanisms difficult [Sal20, LABK17, KRDO17]. Automating this implementation step could prevent bugs in staking protocols and the massive loss of funds that comes with these exploits. Information flow control (IFC) kernels give developers the ability to enforce a variety of security properties by limiting the information flow between applications. However, correctly implementing these restrictions is prohibitively difficult. Existing work enables push-button verification of these security properties for IFC kernels [SNCK+18], but still requires a manually-written implementation. By synthesizing security monitors for IFC kernels that guarantee these properties, the high implementation burden for IFC security could be removed entirely. Overall, program synthesis tools can help systems developers build correct, performant software more effectively and easily for a wide spectrum of system domains.

Bibliography

[AHC+16] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. COGENT: Verifying high-assurance file system implementations. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188, Atlanta, GA, April 2016.

[AMSS10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, pages 258–272, Edinburgh, United Kingdom, July 2010.

[AMSS11] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Saarbrücken, Germany, March–April 2011.

[BA06] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 394–403, San Jose, CA, October 2006.

[BA08] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 177–192, San Diego, CA, December 2008.

[BJA+21] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual conference, October 2021.

[BKL+16] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In

Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 83–98, Atlanta, GA, April 2016.

[Bla10] Dion Blazakis. Interpreter exploitation: Pointer inference and JIT spraying. In *Black Hat DC*, Arlington, VA, February 2010.

[BT17] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 467–481, Barcelona, Spain, June 2017.

[BTGC16] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 775–788, St. Petersburg, FL, January 2016.

[CCK+13] Haogang Chen, Cody Cutler, Taesoo Kim, Yandong Mao, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Security bugs in embedded interpreters. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, Singapore, July 2013. 6 pages.

[CCK+17] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 270–286, Shanghai, China, October 2017.

[CZC+15] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–37, Monterey, CA, October 2015.

[DF84] Jack W. Davidson and Christopher W. Fraser. Automatic generation of peephole optimizations. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 111–116, Montreal, Canada, June 1984.

[dMKA+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover. In *Proceedings of the 25th International Conference on Automated Deduction (CADE)*, pages 378–388, Berlin, Germany, August 2015.

[Edg12] Jake Edge. A library for seccomp filters, April 2012. `https://lwn.net/Articles/494252/`.

[Eng96] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the 17th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 160–170, Philadephia, PA, May 1996.

[FFF+18] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, March 2018.

[Fle17] Matt Fleming. A thorough introduction to eBPF, December 2017. `https://lwn.net/Articles/740157/`.

[FMK+07] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 307–320, Stevenson, WA, October 2007.

[GAG+19] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1069–1084, Phoenix, AZ, June 2019.

[GJTV11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 62–73, San Jose, CA, June 2011.

[GP94] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 49–60, Monterey, CA, November 1994.

[Hen07] Valerie Henson. The many faces of fsck, September 2007.

[Hor18] Jann Horn. Issue 1454: arbitrary read+write via incorrect range tracking in eBPF. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1454`, January 2018.

[Hor19] Jann Horn. libseccomp: incorrect compilation of arithmetic comparisons. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1769`, March 2019.

[HSF21] Fei He, Zhihang Sun, and Hongyu Fan. Satisfiability modulo ordering consistency theory for multi-threaded program verification. In *Proceedings of the 42nd ACM*

SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021, page 1264–1279, New York, NY, USA, 2021. Association for Computing Machinery.

[JNR02] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. In *Proceedings of the 23rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 304–314, Berlin, Germany, June 2002.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, October 2009.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 19–37, San Francisco, CA, May 2019.

[KRDO17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 357–388, Cham, 2017. Springer International Publishing.

[LABK17] Wenting Li, Sébastien Andreina, Jens-Matthias Bohli, and Ghassan Karame. Securing proof-of-stake blockchain protocols. In Joaquin Garcia-Alfaro, Guillermo Navarro-Arribas, Hannes Hartenstein, and Jordi Herrera-Joancomartí, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 297–315, Cham, 2017. Springer International Publishing.

[LADADL13] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–44, San Jose, CA, February 2013.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[LHL19] Juneyoung Lee, Chung-Kil Hur, , and Nuno P. Lopes. AliveInLean: A verified LLVM peephole optimization verifier. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV)*, pages 445–455, New York, NY, July 2019.

[LMNR15] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 22–32, Portland, OR, June 2015.

[LPE+23] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. Chipmunk: Investigating crash-consistency in persistent-memory file systems. In *Proceedings of the 18th ACM EuroSys Conference*, Rome, Italy, May 2023.

[Mas87] Henry Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–126, Palo Alto, CA, October 1987.

[MJ93] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 259–270, San Diego, CA, January 1993.

[MMP+18] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–50, Carlsbad, CA, October 2018.

[MTDC19] Michaël Marcozzi, Qiyi Tang, Alastair Donaldson, and Cristian Cadar. Compiler fuzzing: How much does it matter? In *Proceedings of the 34th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Athens, Greece, October 2019.

[Myr11] Magnus O. Myreen. Verified just-in-time compiler on x86. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL)*, page 107–118, New York, NY, USA, January 2011. Association for Computing Machinery.

[NBG+19] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 225–242, Huntsville, Ontario, Canada, October 2019.

[Nel19] Luke Nelson. bpf, riscv: clear high 32 bits for ALU32 add/sub/neg/lsh/rsh/arsh, May 2019. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1e692f09e091`.

[NPB15] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 9:53–58, 2015.

[OCGO96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[PAA05] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 105–120, Anaheim, CA, April 2005.

[Pau20] Manfred Paul. CVE-2020-8835: Linux kernel privilege escalation via improper eBPF program verification. `https://www.thezdi.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification`, April 2020.

[PCA+14] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448, Broomfield, CO, October 2014.

[PJS+14] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 396–407, Edinburgh, United Kingdom, June 2014.

[RBM13] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage*, 9(3):9:1–32, August 2013.

[RIS19] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 2019121*, December 2019.

[RO91] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Pacific Grove, CA, October 1991.

[Sal20] Fahad Saleh. Blockchain without Waste: Proof-of-Stake. *The Review of Financial Studies*, 34(3):1156–1190, 07 2020.

[SBTW16] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, November 2016.

[SCC+17] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. `https://arxiv.org/abs/1711.04422`, November 2017.

[SiF18] SiFive. SiFive FU540-C000 manual, v1p0. `https://www.sifive.com/boards/hifive-unleashed`, April 2018.

[SLTB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415, San Jose, CA, October 2006.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for c. In *Proceedings of the 13th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Lisbon, Portugal, September 2005.

[SNCK+18] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A framework for design and verification of information flow control systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 287–306, Carlsbad, CA, October 2018.

[Sob15] Louis Sobel. eJitk: Extending Jitk to eBPF. `https://css.csail.mit.edu/6.888/2015/papers/ejitk_sobel.pdf`, May 2015.

[SSA13] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–316, Houston, TX, March 2013.

[TB14] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 530–541, Edinburgh, United Kingdom, June 2014.

[The20] The Coq Development Team. *The Coq Proof Assistant, version 8.12.0*, July 2020.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Braga, Portugal, March–April 2007.

[Twe98] Stephen C. Tweedie. Journaling the Linux ext2fs filesystem. In *Proceedings of the 4th Annual LinuxExpo*, Durham, NC, May 1998.

[URD+13] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. TRANSIT: Specifying protocols with concolic snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 287–296, Seattle, WA, June 2013.

[WBSC17] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*, Paris, France, January 2017.

[WLZ+14] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–47, Broomfield, CO, October 2014.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, San Jose, CA, June 2011.

[YTEM04] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–287, San Francisco, CA, December 2004.

[YTEM06] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. eXplode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, Seattle, WA, November 2006.

[ZSS17] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 347–361, Barcelona, Spain, June 2017.

[ZTH+14] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing databases for fun and profit. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 449–464, Broomfield, CO, October 2014.