

Synthesis-Aided Crash Consistency for Storage Systems

Jacob Van Geffen ✉ 

University of Washington, Seattle, WA, USA

Xi Wang ✉

University of Washington, Seattle, WA, USA

Amazon Web Services, Seattle, WA, USA

Emina Torlak ✉

University of Washington, Seattle, WA, USA

Amazon Web Services, Seattle, WA, USA

James Bornholt ✉ 

The University of Texas at Austin, TX, USA

Amazon Web Services, Seattle, WA, USA

Abstract

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 paper 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.

2012 ACM Subject Classification Software and its engineering → Search-based software engineering; Computer systems organization → Secondary storage organization

Keywords and phrases program synthesis, crash consistency, file systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.35

Funding This material is based upon work supported by the National Science Foundation under Grant No. 2124044.

1 Introduction

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 [34, 22, 35], due to performance optimizations



© Jacob Van Geffen, Xi Wang, Emina Torlak, and James Bornholt;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 35; pp. 35:1–35:26

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 [23], copy-on-write data structures [24], and soft updates [11]. 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 [30]. These optimizations and their interactions are subtle, and have led to severe crash-consistency bugs in well-tested storage systems [17, 7]. 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 [18, 34] and verification [8, 27] tools for validating crash consistency, these tools do not alleviate the burden of implementing crash-consistent systems.

This paper 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 [2, 5] 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 [4], 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 paper 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 paper is organized as follows. Section 2 gives a walk-through of building a simple storage system with DepSynth. Section 3 defines the DepSynth programming model, including labeled writes and dependency rules. Section 4 describes the DepSynth synthesis algorithm for inferring dependency rules, and Section 5 details DepSynth’s implementation in Rosette. Section 6 evaluates the effectiveness of DepSynth. Section 7 discusses related work, and Section 8 concludes.

2 Overview

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.

2.1 Log-structured storage systems

A log-structured storage system persists user data in a sequential log on disk [25]. 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 Figure 2a: 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

```

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

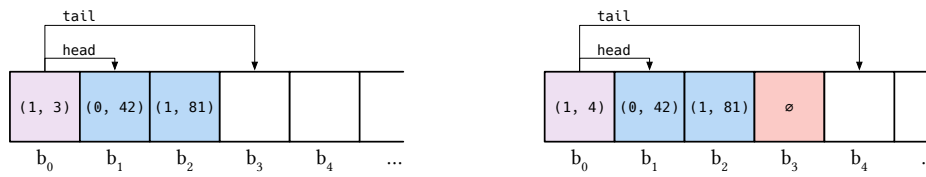
```

■ **Figure 1** Implementation of a simple log-structured key-value store.

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 as shown in Figure 1.

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.

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.



(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 2** The on-disk layout of a simple key-value store. Arrows denote pointers and boxes are blocks.

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 Figure 2b shows. This would leave the `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.

2.2 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 sufficient 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):

35:6 Synthesis-Aided Crash Consistency for Storage Systems

```
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 [2, 32], and have also been used to communicate crash consistency models [5]. 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 [2, 6], 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 [1].

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 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 4 then presents the DepSynth synthesis algorithm for inferring sufficient dependency rules to make a storage system crash consistent.

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:

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

We assume that single-block write operations are atomic, as in previous work [27, 8]. 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.

3.1.1 Dependency rules

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

► **Definition 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 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 $d.\text{write}(a_1, s_1, l_1)$ and $d.\text{write}(a_2, s_2, l_2)$, 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 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.

3.1.2 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 [10] and ShardStore [4], 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 paper 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 2).

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.

3.2.1 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** (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 $I_{O_i}(d, \mathbf{x})$ that takes as input a disk state d and a vector of other inputs \mathbf{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 paper, we use Rosette [28] as our symbolic evaluator; this requires implementation functions to be written in Racket and allows them to be 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 $[O_1(x_1), \dots, O_n(x_n)]$ 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 $I_{O_i}(d, x_i)$ 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.

3.2.2 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 [2, 32], and have also been used to communicate crash consistency [5] and to search for crash consistency bugs in storage systems [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).

3.2.3 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 [12]. The crash-consistency predicate can also include stronger properties such as checking the atomic-replace-via-rename property for POSIX file systems [5, 22].

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 [27]. We first encode the behavior of a single write operation, and then extend that encoding to executions of entire programs.

3.3.1 Write operations

We model the behavior of a disk write operation as a transition function $f_{\text{write}}(d, a, v, s)$, 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_{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:

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

3.3.2 Program executions

Given the trace of write operations $[w_1, \dots, w_n] = \text{Evaluate}_{\mathcal{O}}(P)$ executed by a program P against storage system \mathcal{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:

$$\begin{aligned} & \text{Run}([\text{write}(a_1, v_1, l_1), w_2, \dots, w_n], [s_1, \dots, s_n], d) \\ &= \text{Run}([w_2, \dots, w_n], [s_2, \dots, s_n], f_{\text{write}}(d, a_1, v_1, s_1)) \\ & \quad \text{Run}([], [], d) = d \end{aligned}$$

We call the vector $\mathbf{s} = [s_1, \dots, s_n]$ 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 4** (Valid crash schedule). *Let $[w_1, \dots, w_n] = \text{Evaluate}_{\mathcal{O}}(P)$ be the trace of operations executed by a program P on storage system \mathcal{O} , R be a set of dependency rules, and $\mathbf{s} = [s_1, \dots, s_n]$ the crash schedule for the trace. The crash schedule \mathbf{s} is valid for the program P and set of rules R , written $\text{Valid}_R(\mathbf{s}, P)$, if for all operations $w_i = \text{write}(a_i, v_i, l_i)$ and $w_j = \text{write}(a_j, v_j, l_j)$, whenever there exists a rule $n_a \rightsquigarrow_p n_b \in R$ that matches l_i and l_j , then $s_i \rightarrow s_j$.*

This definition is a logical encoding of dependency safety (Definition 2): if $s_i \rightarrow s_j$, 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 $T = \langle P_{\text{initial}}, P_{\text{main}} \rangle$ as a function of a set of dependency rules R :

► **Definition 5** (Single-test crash consistency). *Let $T = \langle P_{\text{initial}}, P_{\text{main}} \rangle$ be a litmus test. Let $d_{\text{initial}} = \text{Run}(\text{Evaluate}_{\mathcal{O}}(P_{\text{initial}}), \top, d_0)$ be the disk state reached by running the program P_{initial} against storage system \mathcal{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, $\text{Consistent}(\text{Run}(\text{Evaluate}_{\mathcal{O}}(P_{\text{main}}), \mathbf{s}, d_{\text{initial}}))$ holds.*

► **Example 6.** Consider the `SingleEntry_TwoAppend` litmus test from Section 2. Interpreting the initial and main programs gives two traces:

$Interpret(P_{initial}) = [\text{write}(1, \text{to_block}((0, 42)), \langle \text{log}, 0 \rangle),$
 $\text{write}(0, \text{to_block}((1, 2)), \langle \text{superblock}, 0 \rangle)]$

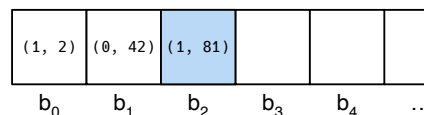
$Interpret(P_{main}) = [\text{write}(2, \text{to_block}((1, 81)), \langle \text{log}, 1 \rangle),$
 $\text{write}(0, \text{to_block}((1, 3)), \langle \text{superblock}, 1 \rangle),$
 $\text{write}(3, \text{to_block}((2, 37)), \langle \text{log}, 2 \rangle),$
 $\text{write}(0, \text{to_block}((1, 4)), \langle \text{superblock}, 2 \rangle)]$

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 2 restricts the valid crash schedules (Definition 4):

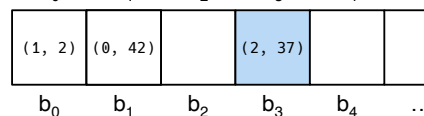
- $\text{superblock} \rightsquigarrow_{=} \text{log}$ requires $s_2 \rightarrow s_1$ and $s_4 \rightarrow s_3$.
- $\text{superblock} \rightsquigarrow_{>} \text{superblock}$ requires $s_4 \rightarrow s_2$.

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

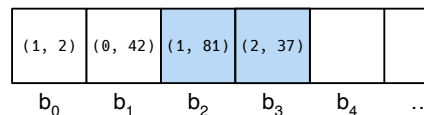
- (1) $[s_1 = \top, s_2 = \perp, s_3 = \perp, s_4 = \perp]$
 (only the first log block is on disk)



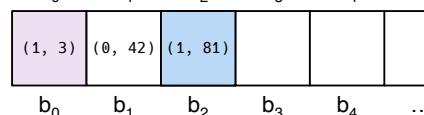
- (2) $[s_1 = \perp, s_2 = \perp, s_3 = \top, s_4 = \perp]$
 (only the second log block is on disk)



- (3) $[s_1 = \top, s_2 = \perp, s_3 = \top, s_4 = \perp]$
 (both log blocks are on disk)



- (4) $[s_1 = \top, s_2 = \top, s_3 = \perp, s_4 = \perp]$
 (the first log block and first superblock write are on disk)



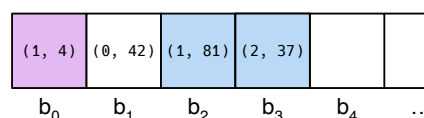
- (5) $[s_1 = \top, s_2 = \top, s_3 = \top, s_4 = \perp]$
 (the first log block, first superblock write, and second log block are on disk)



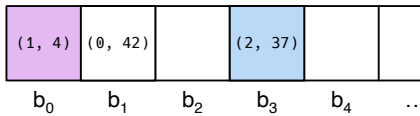
Each of these states satisfies the key-value store’s crash-consistency predicate $Consistent(d)$ defined in Section 2, as in each case the superblock’s `head` and `tail` pointers refer only to log blocks that are also on disk. Some states result in data loss after the crash – for example, neither key 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 5.

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

- (6) $[s_1 = \top, s_2 = \perp, s_3 = \top, s_4 = \top]$
 (the first log block, second log block, and second superblock write are on disk)



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

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 (Figure 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 5). While Definition 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 $Evaluate_{\mathcal{O}}(P) = [\text{write}(a_1, v_1, \langle n_1, t_1 \rangle), \text{write}(a_2, v_2, \langle n_2, t_2 \rangle)]$. 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 $s = \perp$ schedules, as Definition 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 [14] from distributed systems and memory consistency, and require the set of synthesized dependency rules R to be *acyclic*.

4.2 The `DepSynth` Algorithm

The `DEPSYNTH` algorithm (Figure 3) 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 Figure 3), invokes the procedure `RULESFORTTEST` (Section 4.3) to synthesize dependency rules that make T crash consistent, and adds the new rules to the candidate set (line 11). Working incrementally reduces the number of litmus tests for which `DEPSYNTH` needs to synthesize rules; for example, in Section 6.1 we show that only 10 of 16,250 tests were passed to `RULESFORTTEST` 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( $\mathcal{O}$ ,  $\mathcal{T}$ , Consistent)
2    $R \leftarrow \{\}$ 
3   loop
4      $T \leftarrow \text{NEXTTEST}(\mathcal{T}, R, \mathcal{O}, \textit{Consistent})$ 
5     if  $T = \perp$  then  $\triangleright R$  makes all tests in  $\mathcal{T}$  crash consistent
6       return  $R$ 
7      $\mathcal{T} \leftarrow \mathcal{T} \setminus T$ 
8      $R' \leftarrow \text{RULESFORTEST}(T, \mathcal{O}, \textit{Consistent})$ 
9     if  $R' = \perp$  then  $\triangleright$  No rules can make  $T$  crash consistent
10      return UNSAT
11      $R \leftarrow R \cup R'$ 
12     if  $\neg \text{ACYCLIC}(R)$  then  $\triangleright$  Fail if new rules create a cycle in the rule set
13       return UNKNOWN
14 function NEXTTEST( $\mathcal{T}$ ,  $R$ ,  $\mathcal{O}$ , Consistent)
15   for  $T \in \mathcal{T}$  do
16     if  $\neg \text{CRASHCONSISTENT}(T, R, \mathcal{O}, \textit{Consistent})$  then
17       return  $T$ 
18   return  $\perp$ 
 $\triangleright$  Check Def. 5 with an SMT solver
19 function CRASHCONSISTENT( $T = \langle P_{\textit{initial}}, P_{\textit{main}} \rangle$ ,  $R$ ,  $\mathcal{O}$ , Consistent)
20    $d_{\textit{initial}} \leftarrow \text{Run}(\text{Evaluate}_{\mathcal{O}}(P_{\textit{initial}}), \top, d_0)$ 
21   return  $\forall s. \text{Valid}_R(s, P_{\textit{main}}) \Rightarrow \textit{Consistent}(\text{Run}(\text{Evaluate}_{\mathcal{O}}(P_{\textit{main}}), s, d_{\textit{initial}}))$ 

```

■ **Figure 3** 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 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 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 4.1). The algorithm in Figure 3 returns UNKNOWN if such a cycle is found. We have not seen this failure mode occur for the storage systems we evaluated (Section 6), but it is possible in principle. In Section 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 19), 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 5. This procedure uses symbolic evaluation of the storage system implementation \mathcal{O} to generate the logical encoding described in Section 3.3, and solves the resulting formulas using an off-the-shelf SMT solver [20].

4.3 Synthesizing Dependency Rules with Happens-Before Graphs

The core of the DEPSYNTH algorithm is the RULESFORTEST procedure in Figure 4, 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* [14] 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

```

22 function RULESFORTEST( $T = \langle P_{initial}, P_{main} \rangle, \mathcal{O}, Consistent$ )
23    $W \leftarrow \{w \mid w \in Evaluate_{\mathcal{O}}(P_{main})\}$ 
24   return PHASE1( $\mathcal{T}, [], W, \mathcal{O}, Consistent$ )

25 function PHASE1( $T, order, W, \mathcal{O}, Consistent$ )  $\triangleright$  Search for total orders over writes
26   if  $W = \emptyset$  then
27      $G \leftarrow \{(order[i], order[j]) \mid 0 \leq i < j < |order|\}$ 
28     return PHASE2( $\mathcal{T}, G, \mathcal{O}, Consistent$ )  $\triangleright G$  is a total order; minimize it in Phase 2
29   for  $w \in W$  do
30      $order' \leftarrow order + [w]$ 
31      $W' \leftarrow W \setminus \{w\}$ 
32      $G \leftarrow \{(order[i], order[j]) \mid 0 \leq i < j < |order|\} \cup$ 
33        $\{(w_1, w_2) \mid w_1 \in order \wedge w_2 \in W\} \cup$ 
34        $\{(w_1, w_2) \mid w_1, w_2 \in W\}$ 
35     if  $\neg CRASHCONSISTENT(T, RULESFORGRAPH(G), \mathcal{O}, Consistent)$  then
36       continue
37      $R \leftarrow PHASE1(T, order', W', \mathcal{O}, Consistent)$ 
38     if  $R \neq \perp$  then
39       return  $R$ 
40   return  $\perp$ 

39 function PHASE2( $T, G, \mathcal{O}, Consistent$ )  $\triangleright$  Minimize graph  $G$  by removing individual edges
40    $R \leftarrow RULESFORGRAPH(G)$ 
41   if  $\neg CRASHCONSISTENT(T, R, \mathcal{O}, Consistent)$  then
42     return  $\perp$ 
43   for  $(w_1, w_2) \in G$  do  $\triangleright$  Try removing each edge from  $G$ 
44      $G' \leftarrow G \setminus \{(w_1, w_2)\}$ 
45      $R' \leftarrow PHASE2(T, G', \mathcal{O}, Consistent)$ 
46     if  $R' \neq \perp$  then
47       return  $R'$ 
48   if  $ACYCLIC(R)$  then
49     return  $R$   $\triangleright G$  makes  $T$  crash consistent and no subgraph of  $G$  suffices
50   else
51     return  $\perp$ 

52 function RULESFORGRAPH( $G$ )  $\triangleright$  Generalize a happens-before graph into dependency rules
53    $R \leftarrow \{\}$ 
54   for  $(w_1, w_2) \in G$  do
55      $\langle n_1, t_1 \rangle \leftarrow LABEL(w_1)$   $\triangleright$  Get label  $l_1 = \langle n_1, t_1 \rangle$  for write  $w_1 = write(a_1, s_1, l_1)$ 
56      $\langle n_2, t_2 \rangle \leftarrow LABEL(w_2)$ 
57     if  $t_1 < t_2$  then
58        $R \leftarrow R \cup \{n_2 \rightsquigarrow_{>} n_1\}$   $\triangleright$  Invert order, as a rule  $n_a \rightsquigarrow_p n_b$  says  $n_a$  happens after  $n_b$ 
59     else if  $t_1 = t_2$  then
60        $R \leftarrow R \cup \{n_2 \rightsquigarrow_{=} n_1\}$ 
61     else
62        $R \leftarrow R \cup \{n_2 \rightsquigarrow_{<} n_1\}$ 
63   return  $R$ 

```

■ **Figure 4** 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 removes edges from it until the happens-before graph is minimal.

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. RULESFORTTEST 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 a set of dependency rules.

RULESFORTTEST 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.

RULESFORTTEST builds on the observation that crash consistency (Definition 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 7** (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 5), then no subset $R' \subset R$ can make T crash consistent.*

Proof sketch. If R does not make T crash consistent, there exists a valid crash schedule \mathbf{s} (Definition 4) that does not satisfy the crash consistency predicate *Consistent*. By Definition 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 . ◀

RULESFORTTEST 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.

4.3.1 Total order search

PHASE1 (line 25) 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 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 33) and if so, prunes this branch of the search. For PHASE1 to be complete, this check must behave angelically 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 32). If the test cannot be made crash consistent even with every possible rule included, then by Theorem 7 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.

4.3.2 Partial order search

Starting from a happens-before graph G that reflects a total order over all writes in T , PHASE2 (line 39) 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 44), checks if the graph remains sufficient for crash consistency (line 41), 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 7), it is complete – if an acyclic solution exists, PHASE2 will reach it.

4.3.3 Generating rules from happens-before graphs

The RULESFORTTEST search operates on happens-before graphs, but its goal is to synthesize dependency rules (Definition 1). The RULESFORGRAPH procedure (line 52) 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 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 RULESFORTTESTS to avoid overfitting to the input litmus tests. In Section 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.

4.3.4 Properties of RulesForTest

The RULESFORTTEST 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 4.1. RULESFORTTEST 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, RULESFORTTEST 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 7 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 1, the possible epoch predicates are disjoint), removing any rule would incorrectly allow reordering of its corresponding edge(s).

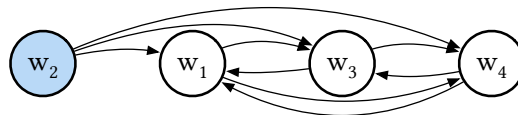
► **Example 8.** Consider running RULESFORTTEST for the simple log-structured key-value store and `SingleEntry_TwoAppend` litmus test from Section 2. From Example 6 we know that this test produces a set W of four writes:

```

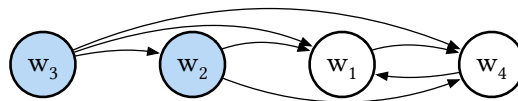
w1 = write(2, to_block((1, 81)), ⟨log, 1⟩),
w2 = write(0, to_block((1, 3)), ⟨superblock, 1⟩),
w3 = write(3, to_block((2, 37)), ⟨log, 2⟩),
w4 = write(0, to_block((1, 4)), ⟨superblock, 2⟩)

```

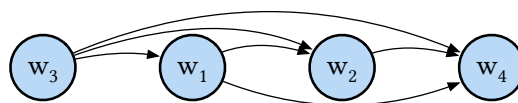
PHASE1 first chooses the first write to add to the total order. Suppose it chooses w_2 . This choice results in the following graph G at line 32 (shaded nodes are in *order*; white nodes are in W):



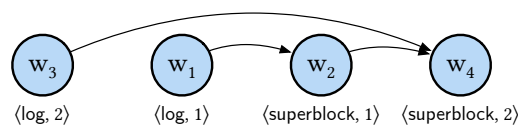
The check at line 33 finds that this graph is not crash consistent: it allows a crash schedule where w_2 is on disk but no other writes are, which violates the crash-consistency predicate as w_2 is a superblock write pointing to a log block that is not on disk. PHASE1 therefore continues (line 34), which 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 33 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 chooses w_2 first, PHASE1 eventually reaches line 28 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 45 on the graph $G' = G \setminus \{(w_3, w_1)\}$. This graph still ensures crash consistency at line 41, 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 43 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 2:

$$\begin{array}{ll} \text{superblock} \rightsquigarrow_{=} \text{log} & \text{from edges } (w_3, w_4) \text{ and } (w_1, w_2) \\ \text{superblock} \rightsquigarrow_{>} \text{superblock} & \text{from edge } (w_2, w_4). \end{array}$$

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 4.1. In practice, we have not seen this happen for the storage systems we evaluate in Section 6, and so the version of DEPSYNTH presented in Figure 3 fails if the synthesized rules contain a cycle.

To handle cyclic rules, RULESFORTTEST 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 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 RULESFORTTEST 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 RULESFORTTEST.

5 Implementation

We implement both the DepSynth algorithm and the storage systems we study in Section 6 in Rosette [28], an extension of Racket [9] 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 Figure 3. 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 [27] or C [19] in which storage systems are more commonly implemented.

5.1 Ordering

The DEPSYNTH algorithm in Figure 3 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 RULESFORTTEST to explore. Similarly, RULESFORTTEST 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. `RULESFORTTEST` therefore prefers to choose writes in `PHASE1` in program order, and prefers to remove edges in `PHASE2` that contradict program order.

5.2 Reducing solver queries

Both `PHASE1` and `PHASE2` in Figure 4 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 `RULESFORTTEST` 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`.

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? (§6.1)
2. Can DepSynth help storage system developers avoid crash-consistency bugs? (§6.2)
3. Does DepSynth’s approach support a variety of storage system designs? (§6.3)

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 [4], the exabyte-scale production storage node for the Amazon S3 object storage service.

6.1.1 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) [21], 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.

6.1.2 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

■ **Table 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.

Test	Test Length	Writes	Allowed by both		Allowed only by DepSynth		Allowed only by ShardStore	
			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 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 RULESFORTTEST procedure (line 8 in Figure 3) only 10 times, showing that DepSynth’s incremental approach is effective at reducing the search space.

6.1.3 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 [11] 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 [10]. We therefore compare our synthesized rules against ShardStore’s dependency graphs to see how well DepSynth may replace 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 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 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.

6.1.4 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 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.

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 [4, 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 2** Sample crash-consistency bugs in three storage systems reported by two recent papers [4, 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	<code>fsync</code> failed after directory rename (ade990f)	Yes
f2fs	Wrong file size when zeroing file beyond EOF (17cd07a)	No – not reordering

Table 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-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.

6.3 Other Storage Systems

Beyond ShardStore, we expect DepSynth to effectively generate rules for any storage systems whose crash consistency properties can be ensured by correctly ordering writes. As Frost et al. describe in [10], write-before relationships underlie *every* crash consistency mechanism, including journaling, synchronous writes, copy-on write data structures, and soft updates. Though storage systems vary greatly in their mechanisms for storing and retrieving data, each must enable crash consistency by enforcing write-before relationships. Since DepSynth is a tool for automatically developing write-before relationships, this leads us to believe that DepSynth can be a useful tool for automating crash consistency in all such systems.

To demonstrate this point, we have also used DepSynth to implement a log-structured file system [25]. 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 `RULESFORTTEST` for only 13 tests. This result shows that DepSynth can automate crash consistency for storage systems other than key-value stores.

7 Related Work

7.1 Verified storage systems

Inspired by successes in other systems verification problems [16, 13], 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 [27] 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 [8] 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 [7] 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 [3] 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.

7.2 Crash-consistency bug-finding tools

Ferrite [5] 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 [18] is a tool for finding crash-consistency bugs in Linux file systems. 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. Chipmunk [15] 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. 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 [33] and eXplode [34] 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. [4] 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 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.

7.3 Program synthesis for systems code

Transit [31] is a tool for automatically inferring distributed protocols such as those used for cache coherence. It guides the search using *concolic* snippets [26] – effectively litmus tests that can be partially symbolic – and finds a protocol that satisfies those snippets for *any* ordering of messages. MemSynth [6] 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 [29] to shrink the search space of happens-before graphs would be promising future work.

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.

References

- 1 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.
- 2 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.
- 3 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.
- 4 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.
- 5 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.

- 6 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.
- 7 Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nikolai 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.
- 8 Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nikolai 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.
- 9 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.
- 10 Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoaka, 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.
- 11 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.
- 12 Valerie Henson. The many faces of fsck, September 2007. URL: <https://lwn.net/Articles/248180/>.
- 13 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.
- 14 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.
- 15 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.
- 16 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.
- 17 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.
- 18 Jayashree Mohan, Ashlie Martinez, Soujanya Ponnappalli, 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.
- 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.
- 20 Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 9:53–58, 2015.
- 21 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.

- 22 Thanumalayan Sankaranarayanan Pillai, Vijay Chidambaram, Ramnathan 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.
- 23 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.
- 24 Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage*, 9(3):9:1–32, August 2013.
- 25 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.
- 26 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.
- 27 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.
- 28 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.
- 29 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.
- 30 Stephen C. Tweedie. Journaling the Linux ext2fs filesystem. In *Proceedings of the 4th Annual LinuxExpo*, Durham, NC, May 1998.
- 31 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.
- 32 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.
- 33 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.
- 34 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.
- 35 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.