Synthesis-Aided Crash Consistency for Storage Systems

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 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 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 a major cloud provider. The evaluation shows that* DEPSYNTH *can automate crash consistency for this complex storage system, with similar results to existing expert-written code, and can identify and correct consistency and performance issues.*

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 [19, 31, 32], 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 [20], copy-on-write data structures [21], and soft updates [10]. However, using these approaches is 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 [27]. These optimizations are subtle and have led to crash-consistency bugs in well-tested storage systems [7, 15]. Second, developers must implement their system using low-level *durability* APIs such as `write` and `fsync` that offer no direct support for enforcing *consistency* properties, and so they are left to roll their own consistency mechanisms. While prior work offers testing [16, 31] and verification [8, 24] tools for validating crash consistency, these tools do not overcome the initial cost of implementing a crash-consistent system.

This paper presents a new *synthesis-aided* programming model for building crash-consistent storage systems. The model consists of three parts: a high-level storage interface based on *labeled writes*; a synthesis engine for turning labeled writes and a crash consistency property into a set of *dependency rules* that writes must follow; and a *dependency-aware buffer cache* that enforces these rules at run time. Together, these three components decouple storage systems from their crash consistency mechanisms. 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 use two writes to append to the journal: one to append the data block to the journal (labeled data) and one to update a superblock to point to the new tail block (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, the developer uses a new program synthesizer, DEPSYNTH, to automatically generate *dependency rules* that writes must follow. A dependency rule uses labels to define an ordering requirement between two writes: writes with one label must persist before 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 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 generate a set of dependency rules that guarantee the crash-consistency predicate for every litmus test. Although this approach is example-guided and so guarantees crash consistency only 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 a production key-value store developed by a major commercial cloud provider.[1] 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;
- DEPSYNTH, a synthesis tool that can infer dependency rules to make a storage system crash consistent; and
- An evaluation showing that DEPSYNTH supports different storage system designs and scales to production systems.

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.

Log-structured storage systems. A log-structured storage system persists user data in a sequential log on disk [22]. 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.

[1]Anonymized for double-blind review.

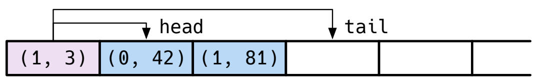
Consider implementing a simple key-value store as a log-structured storage system. The on-disk data structure comprises two parts as shown in Fig. 1a: a log that stores key-value pairs (with one pair per block), and a superblock that holds pointers to the head and tail of the log. We will assume that 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():
      self.head, self.tail = 1, 1
    else:
      self.head, self.tail = from_block(superblock)
    self.epoch = 0

  def put(self, key: int, value: int):
    addr = self.tail
    self.tail += 1

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

    new_sb = to_block(self.head, self.tail)
    disk.write(0, new_sb, ("superblock", self.epoch))

    self.epoch += 1

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

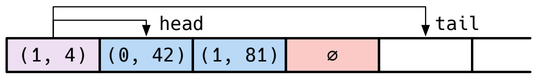
Calls to `disk.read` and `disk.write` illustrate our new higher-level storage interface: `disk.read` is unchanged from the usual system call, taking as input an address on the disk to read from; and `disk.write` takes as input an address on the disk to write to, the block data to write to that address, 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,[2] 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 on a typical 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. 1b shows. This would leave the `tail` pointer referring to an uninitialized

[2]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 1: The on-disk layout of a simple key-value store. Arrows denote pointers and boxes are blocks.

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, the implementation of the storage system. Second, 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 `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.head, self.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, 29], 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 handwritten implementation: while the first rule ensures consistency for a *single* put operation, it still allows `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 `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 with this interface. § 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:

- $d.\texttt{write}(a,v,l)$: write a block v to 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 [8, 24]. These interfaces are similar to standard `pwrite` and `pread` APIs except that `write` takes as input a *label*. 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) inspect to enforce ordering requirements. Although the two components of a label together identify a write, they serve separate purposes: the name indicates which on-disk data structure the write targets, while the epoch associates related writes across different names. Names are strings but are uninterpreted other than to check equality between them. Epochs are integers that serve 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 provides* dependency safety *for a set of dependency rules R if, whenever a 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. § 3.3 will specify dependency safety more formally by defining the crash behavior of a disk in first-order logic.

The epoch predicate scopes a dependency 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 with 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 in 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 (i.e., it enforces dependency safety (Def. 3.2).

The dependency-aware buffer cache is inspired by higher-level storage APIs such as Featherstitch [9] and ShardStore [4], which also provide interfaces for specifying ordering requirements for writes. Unlike these imperative interfaces, the dependency-aware buffer cache is declarative, configuring dependency rules just once and then automatically applying them to every relevant write without requiring the developer to manually construct ordering graphs. Implementation details of a dependency-aware buffer cache are outside the scope of this paper (ours roughly follows ShardStore), but could use a variety of primitives 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 (Def. 3.2).

3.2. Storage Systems and Litmus Tests

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

Storage system implementations. Developers implement a storage system for DEPSYNTH by defining a collection of API operations $\mathcal{O}$ and an implementation 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 *$I_{O_i}(d,x)$ that takes as input a disk state d and a vector of other inputs x and issues write operations to mutate disk d.*

DepSynth needs to be able to symbolically evaluate implementation functions with respect to a symbolic disk state d. In this paper, we use Rosette [25] as our symbolic evaluator; this allows implementation functions to be written in Racket and automatically lifted to support symbolic evaluation, so long as their executions are deterministic and bounded.

We say that a *program* P is a sequence of calls $[O_1(x_1), \ldots, O_n(x_n)]$ to API operations $O_i \in \mathcal{O}$. Given a program P, we write $\mathit{Eval}_{\mathcal{O}}(P)$ 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 $[w_1, \ldots, w_n]$ that the program performed. The trace does not include reads 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 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 or crash consistency models to developers [2, 5, 29] and to search for crash consistency bugs in storage systems [16]. DepSynth is agnostic to the source of the litmus tests it uses so long as they are straight-line and deterministic.

Crash consistency predicates. To define consistency for their system, developers also provide a *crash-consistency predicate* $\mathit{Consistent}(d)$ 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 (no dangling pointers). These properties correspond to those used by an `fsck`-like checker [11]. It can also include stronger properties such as checking the atomic-replace-via-rename property for POSIX file systems [5, 19].

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 [24]. We first encode the behavior of a single write operation, and then 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 $[w_1, \ldots, w_n] = \mathit{Eval}_{\mathcal{O}}(P)$ 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:

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

We call the vector $s = [s_1, \ldots, s_n]$ 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 dependency rules:

Definition 3.4 (Valid crash schedule)**.** *Let* $[w_1, \ldots, w_n] = \mathrm{Eval}_{\mathcal{O}}(P)$ *be the trace of operations executed by a program* P *on storage system* $\mathcal{O}$*,* R *be a set of dependency rules, and* $s = [s_1, \ldots, s_n]$ *the crash schedule for the trace. The crash schedule* s *is* valid *for the program* P *and set of rules* R*, written* $\mathrm{Valid}_R(s, P)$*, 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* $s_i \rightarrow s_j$*.*

This definition is a logical encoding of dependency safety (Def. 3.2): 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 $T = \langle P_{initial}, P_{main} \rangle$ under 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* $d_{\text{initial}} = \mathrm{Run}\left(\mathrm{Eval}_{\mathcal{O}}(P_{\text{initial}}), \top, d_0\right)$ *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* s *such that* $\mathrm{Valid}_R(s, P_{\text{main}})$ *is true,* $\mathrm{Consistent}(\mathrm{Run}\left(\mathrm{Eval}_{\mathcal{O}}(P_{\text{main}}), s, d_{\text{initial}}\right))$ *holds.*

As an example, consider the `SingleEntry_TwoAppend` litmus test from § 2. Interpreting the initial and main programs gives two traces:

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

Let $s = [s_1, s_2, s_3, s_4]$ be a crash schedule for P_{main}. Applying the two synthesized rules from § 2 restricts the valid crash schedules (Def. 3.4):

- superblock $\rightsquigarrow_=$ log requires $s_2 \to s_1$ and $s_4 \to s_3$.
- superblock $\rightsquigarrow_>$ superblock requires $s_4 \to s_2$.

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

1. $[s_1 = \top, s_2 = \bot, s_3 = \bot, s_4 = \bot]$ (first log block on disk)
2. $[s_1 = \bot, s_2 = \bot, s_3 = \top, s_4 = \bot]$ (second log block on disk)
3. $[s_1 = \top, s_2 = \bot, s_3 = \top, s_4 = \bot]$ (both log blocks on disk)
4. $[s_1 = \top, s_2 = \top, s_3 = \bot, s_4 = \bot]$ (first log block and first superblock write on disk)
5. $[s_1 = \top, s_2 = \top, s_3 = \top, s_4 = \bot]$ (first log block, first superblock write, and second log block on disk)

Each of these states satisfies the crash-consistency predicate *Consistent*(d) defined in § 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 Def. 3.5.

4. Dependency Rule Synthesis

This section describes the DEPSYNTH synthesis algorithm for generating dependency rules that guarantee crash consistency for a set of litmus tests. We first formalize the dependency rule synthesis problem and then present the core DEPSYNTH algorithm (Fig. 2).

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 $\mathscr{T}$ crash consistent (Def. 3.5). While solving for the set of rules R in Def. 3.5 would suffice to guarantee crash consistency, it would not prevent *cyclic* solutions that cannot be executed. For example, consider a program P with two writes labelled n_1 and n_2. Then $R = \{n_1 \rightsquigarrow_= n_2, n_2 \rightsquigarrow_= n_1\}$ guarantees crash consistency because the only valid crash schedules are the trivial cases $s = \top$ and $s = \bot$; in effect, these rules require 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 [13] 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 (Fig. 2) takes as input a storage system implementation $\mathscr{O}$, a set of litmus tests $\mathscr{T}$, 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 $\mathscr{T}$ crash consistent.

```
1  function DEPSYNTH($\mathscr{O}$, $\mathscr{T}$, Consistent)
2      $R \leftarrow \{\}$
3      loop
4          $T \leftarrow$ NEXTTEST($\mathscr{T}, R, \mathscr{O}, Consistent$)
5          if $T = \bot$:              ▷ R makes all tests in $\mathscr{T}$ crash consistent
6              return $R$
7          $\mathscr{T} \leftarrow \mathscr{T} \setminus T$
8          $R' \leftarrow$ RULESFORTEST($T, \mathscr{O}, Consistent$)
9          if $R' = \bot$:             ▷ No rules can make T crash consistent
10             return UNSAT
11         $R \leftarrow R \cup R'$
12         if $\neg$ACYCLIC($R$):
13             return UNKNOWN

14 function NEXTTEST($\mathscr{T}$, R, $\mathscr{O}$, Consistent)
15     for $T \in \mathscr{T}$:
16         if $\neg$CRASHCONS($T, R, \mathscr{O}, Consistent$):
17             return $T$
18     return $\bot$

   ▷ Check Def. 3.5 with an SMT solver
19 function CRASHCONS($T = \langle P_{init}, P_{main} \rangle$, R, $\mathscr{O}$, Consistent)
20     $d_{init} \leftarrow Run(Eval_{\mathscr{O}}(P_{init}), \top, d_0)$
21     return $\forall s.\ Valid_R(s, P_{main}) \Rightarrow Consistent(Run(Eval_{\mathscr{O}}(P_{main}), s, d_{init}))$
```

Figure 2: The DEPSYNTH algorithm takes as input a storage system implementation $\mathscr{O}$, a set of litmus tests $\mathscr{T}$, and a crash-consistency predicate *Consistent*, and returns an acyclic set of dependency rules that make all tests in $\mathscr{T}$ crash consistent.

DEPSYNTH does not try to generate a set of dependency rules for all tests in $\mathscr{T}$ at once, as 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. 2), invokes the procedure RULESFORTEST (§ 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 § 5.1 we show that only 10 of 16,250 tests were used by RULESFORTEST to synthesize 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.

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 (§ 4.1). The algorithm in Fig. 2 returns UNKNOWN if such a cycle is found. We have not seen this occur for the storage systems we evaluated (§ 5), but it is possible in principle. In § 4.4, we outline how our implementation extends Fig. 2 to recover from cycles by generalizing RULESFORTEST to solve for multiple tests at once.

DEPSYNTH delegates checking for crash consistency to the procedure CRASHCONS (line 18), 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 Def. 3.5. This procedure uses symbolic evaluation of the storage system implementation $\mathcal{O}$ to generate the logical encoding described in § 3.3, and solves the resulting formulas using an off-the-shelf SMT solver [17].

4.3. Synthesizing Rules with Happens-Before Graphs

The core of the DEPSYNTH algorithm is the RULESFORTEST procedure in Fig. 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* [13] 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 suffices to ensure crash consistency for T, and then syntactically generalizes that happens-before graph into 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. Crash consistency is monotonic over sets of dependency rules—if a set of rules R is not sufficient, then no subset of R is sufficient either—and so RULESFORTEST prunes the search space by checking crash consistency for a happens-before graph G before exploring any subgraphs of G.

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 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 32). If the test cannot be made crash consistent even with every

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

25 function PHASE1($T$, $order$, $W$, $\mathcal{O}$, $Consistent$)
26     if $W = \emptyset$:                                        ▷ $G$ is a total order
27         $G \leftarrow \{(order[i], order[j]) \mid 0 \leq i < j < |order|\}$
28         return PHASE2($\mathcal{T}$, $G$, $\mathcal{O}$, $Consistent$)
29     for $w \in W$:
30         $order' \leftarrow order + [w]$
31         $W' \leftarrow W \setminus \{w\}$
32         $G \leftarrow \{(order[i], order[j]) \mid 0 \leq i < j < |order|\} \cup$
                   $\{(w_1, w_2) \mid w_1 \in order \wedge w_2 \in W\} \cup$
                   $\{(w_1, w_2) \mid w_1, w_2 \in W\}$
33         if $\neg$CRASHCONS($T$, RULESFORGRAPH($G$), $\mathcal{O}$, $Consistent$):
34             continue
35         $R \leftarrow$ PHASE1($T$, $order'$, $W'$, $\mathcal{O}$, $Consistent$)
36         if $R \neq \bot$:
37             return $R$
38     return $\bot$

39 function PHASE2($T$, $G$, $\mathcal{O}$, $Consistent$)
40     $R \leftarrow$ RULESFORGRAPH($G$)
41     if $\neg$CRASHCONS($T$, $R$, $\mathcal{O}$, $Consistent$):
42         return $\bot$
43     for $(w_1, w_2) \in G$:                      ▷ 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 $R' \neq \bot$:
47             return $R'$
48     if ACYCLIC($R$):      ▷ $G$ makes $T$ consistent and no subgraph suffices
49         return $R$
50     else
51         return $\bot$

52 function RULESFORGRAPH($G$)
53     $R \leftarrow \{\}$
54     for $(w_1, w_2) \in G$:
55         $\langle n_1, t_1 \rangle \leftarrow$ LABEL($w_1$)
56         $\langle n_2, t_2 \rangle \leftarrow$ LABEL($w_2$)
57         if $t_1 < t_2$:
58             $R \leftarrow R \cup \{n_2 \rightsquigarrow_{>} n_1\}$
59         else if $t_1 = t_2$:
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 3: The algorithm for generating sufficient dependency rules for a litmus test T searches happens-before graphs over the writes performed by T. PHASE1 searches for total orders over the writes that suffice for crash consistency. Once such a total order is found, PHASE2 removes edges from it until the happens-before graph is minimal.

possible rule included, no subset of those rules (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.

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 it is complete—if an acyclic solution exists, PHASE2 will reach it.

Generating rules from happens-before graphs. As the final step of RULESFORTEST, the RULESFORGRAPH procedure (line 52) takes as input a happens-before graph G and returns a set of dependency rules R that enforce the ordering requirements G requires. 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.

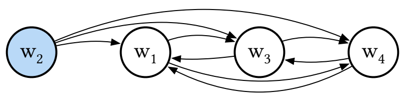
This approach can lead to rules that are too general; for example, it may generate a rule that applies to all epochs when G only required an ordering for *some* epochs. Overly general rules could reduce performance by preventing reordering optimizations that would be safe. However, this same generality also allows RULESFORTESTS to avoid overfitting to the input litmus tests. In § 5.1 we show that generated rules generalize well in practice (i.e., do not overfit), and that they filter out few additional schedules compared to expert-written rules.

Properties of RULESFORTEST. The RULESFORTEST algorithm is *sound*: all returns are guarded by checks of crash consistency and of acyclicity, and so satisfy the requirements of § 4.1. RULESFORTEST is also *complete*: each of PHASE1 and PHASE2 are complete and so together search every total order. Every possible acyclic solution is a subgraph of some total order, since the transitive closure of edges in a happens-before graph is a (strict) partial order, and so exploring all total orders can reach any possible acyclic solution. Finally, RULESFORTEST is *minimal*: removing any rule from a returned set R would violate crash consistency. PHASE2 continues removing edges from a candidate graph G until it cannot be made smaller, and therefore finds a minimal happens-before graph. Every rule in R is justified by an edge in G, and since dependency rules cannot overlap (the epoch predicates in Def. 3.1 are disjoint), removing a rule would incorrectly allow reordering of its corresponding edge(s).

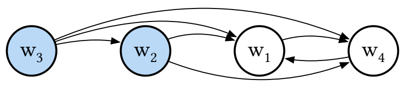
Example. Consider running RULESFORTEST for the log-structured key-value store and `SingleEntry_TwoAppend` litmus test from § 2. From § 3.3 we know that this test produces a set W of four writes:

$$w_1 = \texttt{write}(2, \texttt{to\_block}((1,81)), \langle \mathsf{log}, 1 \rangle),$$
$$w_2 = \texttt{write}(0, \texttt{to\_block}((1,3)), \langle \mathsf{superblock}, 1 \rangle),$$
$$w_3 = \texttt{write}(3, \texttt{to\_block}((2,37)), \langle \mathsf{log}, 2 \rangle),$$
$$w_4 = \texttt{write}(0, \texttt{to\_block}((1,4)), \langle \mathsf{superblock}, 2 \rangle)$$

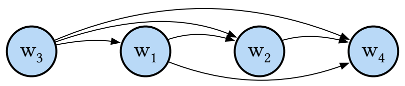
PHASE1 first chooses a write to add to the partial order. Suppose it chooses w_2. This choice gives the following graph G at line 32 (shaded nodes are in *order*; white nodes are in W):



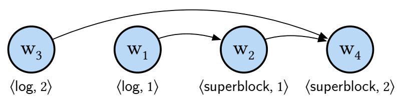
The check at line 33 finds this graph is not crash consistent: it allows a crash schedule where w_2 is on disk but no other writes are, which violates the crash-consistency predicate as w_2 is a superblock write pointing to a log block that is not on disk. PHASE1 thus continues (line 34) by ruling out any total order that starts with $[w_2]$, 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 choose w_2 to add to the total order, giving this graph G:



Again, line 33 finds this graph is not crash consistent—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. 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 § 2:

superblock $\rightsquigarrow_{=}$ log from edges (w_3, w_4) and (w_1, w_2)
superblock $\rightsquigarrow_{>}$ superblock from edge (w_2, w_4).

4.4. Resolving Cycles in Dependency Rules

The top-level DEPSYNTH algorithm generates rules for each litmus test independently, but the union of these rules could be cyclic. Our implementation resolves cycles by extending Fig. 2 to support synthesizing rules for multiple litmus tests at once. It 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 minimizes that entire ordering in PHASE2. Edges between writes from *different* tests have no influence on crash consistency and so will eventually be removed in 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 we could just invoke RULESFORTEST with the entire input set $\mathscr{T}$, but this would be prohibitively expensive. Instead, our implementation resolves cycles in DEPSYNTH by identifying which litmus tests the rules in the cycle were generated from and passes only that subset of tests to the extended RULESFORTEST.

5. Evaluation

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

1. Can developers use DEPSYNTH to synthesize dependency rules for a realistic storage system rather than implementing their own crash-consistency approach by hand? (§5.1)
2. Can DEPSYNTH help storage system developers avoid crash-consistency bugs? (§5.2)
3. Does DEPSYNTH's approach support a variety of storage system designs? (§5.3)

5.1. CLOUDKV 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 CLOUDKV[3] [3], a recently published production key-value store for a major cloud storage service.

Implementation. The first step in using DEPSYNTH is to implement the storage system itself. CLOUDKV is a log-structured merge tree (LSM tree) [18] but with values stored outside the tree. Our CLOUDKV-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 CLOUDKV 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 an implementation in hand, a developer can use DEPSYNTH to synthesize dependency rules that make the system crash consistent. For litmus tests, we randomly generated 16,250 tests 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 CLOUDKV in 49 minutes. To find a correct solution for all 16,250 litmus tests, the DEPSYNTH algorithm invoked the RULESFORTEST procedure (line 8 in Fig. 2) only 10 times, showing that DEPSYNTH's incremental approach is effective at reducing the search space.

Comparison to an existing implementation. CLOUDKV is an existing production system and already supports crash consistency. It does not use declarative dependency rules like DEPSYNTH. Instead, it implements a soft-updates approach [10] that constructs dependency graphs at run time and sequences writes to disk based on those graphs, similar to patchgroups in Featherstitch [9]. We therefore compare our synthesized rules against CLOUDKV'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 CLOUDKV, we used an SMT solver to compute the set of valid crash schedules (Def. 3.4) according to those rules. We then executed the test using the production CLOUDKV 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 a correctness issue in the synthesized rules (allowing schedules that are not crash consistent) or a performance issue in CLOUDKV (forbidding schedules that are safe). We found that every schedule allowed by DEPSYNTH is crash consistent, and that CLOUDKV inserts unnecessary edges in its dependency graphs, ruling out some reorderings that

[3]Anonymized for double-blind review.

Table 1: Schedules allowed by the CloudKV service versus the synthesized dependency rules. A schedule allowed only by one system means either the system 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 CloudKV (one consistency, one performance) that we identified by manually inspecting the "Original" schedules.

			Allowed by both		Allowed only by DepSynth		Allowed only by CloudKV	
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

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, CloudKV 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, CloudKV engineers favored a stronger, simpler invariant in these cases, where DepSynth is able to identify opportunities for performance improvements.

2. Schedules allowed only by CloudKV mean either a correctness issue in CloudKV (allowing schedules that are not crash consistent) or a performance issue in the synthesized rules (forbidding schedules that are safe). 67% of these schedules are incorrectly allowed by CloudKV due to a rare crash-consistency issue that was independently discovered concurrently with this work. We have confirmed with CloudKV engineers that the issue was an 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. After fixing this issue we manually inspected the remaining schedules 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 CloudKV reimplementation agrees that all the resulting states are consistent.

After fixing the two CloudKV issues discussed above, the synthesized dependency rules agree with CloudKV 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, these results show that DepSynth achieves similar results to an expert-written crash consistency implementation, and can identify correctness and performance issues in existing systems.

Generalization. One risk for example-guided synthesis techniques like DepSynth is that they can overfit to the examples (litmus tests) and so be incorrect on unseen test cases. To test generalization, we randomly generated an additional 136,000 litmus tests for our CloudKV-like system. We allowed these tests to be longer (up to 40 writes) than the tests used during synthesis (up to 20 writes). 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 satisfied the crash consistency predicate.

5.2. 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, 16] in three production storage systems for which a known fix is available. We manually analyze each bug and determine whether DepSynth could prevent them.

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. 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 trusts this predicate to be correct. Another ShardStore bug involved a collision between two randomly generated UUIDs. While in principle a litmus test approach could find this bug, 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.

5.3. Other Storage Systems

In addition to the CloudKV case study, we have also used DepSynth to implement a log-structured file system [22]. The file system supports five standard POSIX operations:

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

Table 2: Sample crash-consistency bugs in three storage systems reported by two recent papers [4, 16]. Each bug includes its identifier (bug number for ShardStore, kernel Git commit for btrfs and f2fs).

`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.

6. Related Work

Verified storage systems. Inspired by successes in other systems verification problems [12, 14], 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 [24] 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.

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 [16] 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. This exhaustive test generation is a potential source of litmus tests for DEPSYNTH. FiSC [30] and eXplode [31] use model checking to find bugs in storage systems.

Program synthesis for systems code. Transit [28] is a tool for automatically inferring distributed protocols such as those used for cache coherence. It guides the search using *concolic* snippets [23]—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 [26] to shrink the search space of happens-before graphs would be promising future work.

7. 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 cost 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] Anonymous. Anonymized for double blind review. 2022.

[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 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.