JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Specifying and Checking File System Crash-Consistency Models

Steven Lang

September 4, 2016

Motivation

The problem:

- POSIX file-system-interfaces do not define possible outcomes of a crash
- Can lead to
  - Corrupt application states
  - Catastrophic data loss

Motivation

Replace-Via-Rename Pattern

```
/* "file" has old data */
fd = open("file.tmp");
write(fd, new, size);
close(fd);
rename("file.tmp", "file");
```

Motivation

Replace-Via-Rename Pattern

```
/* "file" has old data */
fd = open("file.tmp");
write(fd, new, size);
close(fd);
rename("file.tmp", "file");
```

$$\begin{tikzcd}
\texttt{open} \arrow[dd] \arrow[dr, bend left] & \\
& \texttt{write}_0 \arrow[d] \\
\texttt{rename} & \texttt{write}_1
\end{tikzcd}$$

Motivation

Replace-Via-Rename Pattern

```
/* "file" has old data */
fd = open("file.tmp");
write(fd, new, size);
close(fd);
rename("file.tmp", "file");
```

$$\begin{tikzcd} \texttt{open} \arrow[d] \arrow[dr, bend left] & \\ & \texttt{write}_0 \arrow[d] \\ \texttt{rename} & \texttt{write}_1 \end{tikzcd}$$

file's on-disk state	possible executions seen on disk
new	open, $\texttt{write}_0$, rename, $\texttt{write}_1$, ...

Motivation

Replace-Via-Rename Pattern

```
/* "file" has old data */
fd = open("file.tmp");
write(fd, new, size);
close(fd);
rename("file.tmp", "file");
```

$$
\begin{tikzcd}
\texttt{open} \arrow[dd] \arrow[dr, bend left] & \\
& \texttt{write}_0 \arrow[d] \\
\texttt{rename} & \texttt{write}_1
\end{tikzcd}
$$

file's on-disk state	possible executions seen on disk
new	open, $\texttt{write}_0$, rename, $\texttt{write}_1$, ...
old	open, $\texttt{write}_0$, crash

Motivation

Replace-Via-Rename Pattern

```
/* "file" has old data */
fd = open("file.tmp");
write(fd, new, size);
close(fd);
rename("file.tmp", "file");
```

$$
\begin{tikzcd}
\texttt{open} \arrow[dd] \arrow[dr, bend left] & \\
& \texttt{write}_0 \arrow[d] \\
\texttt{rename} & \texttt{write}_1
\end{tikzcd}
$$

file's on-disk state	possible executions seen on disk
new	open, $\texttt{write}_0$, rename, $\texttt{write}_1$, ...
old	open, $\texttt{write}_0$, crash
empty	open, rename, crash

Motivation

Replace-Via-Rename Pattern

```
/* "file" has old data */
fd = open("file.tmp");
write(fd, new, size);
close(fd);
rename("file.tmp", "file");
```

$$\begin{tikzcd} \texttt{open} \arrow[dd] \arrow[dr, bend left] & \\ & \texttt{write}_0 \arrow[d] \\ \texttt{rename} & \texttt{write}_1 \end{tikzcd}$$

file's on-disk state	possible executions seen on disk
new	open, $\texttt{write}_0$, rename, $\texttt{write}_1$, ...
old	open, $\texttt{write}_0$, crash
empty	open, rename, crash
partial new	open, $\texttt{write}_0$, rename, crash

Introduction

- Modern file system optimizations relax the order in which operations are executed

Good:

- Provide significant performance gains

Bad:

- Invisible to applications
- Machine crashes during out-of-order execution can harm the data's persistence

Introduction

- Key challenge for application writers:
  - Understand the behavior of file systems across system crashes
- They make assumptions about crash guarantees provided by file systems
- They base their aplications on these assumptions

Introduction

- Being too optimistic about crash guarantees leads to serious **data losses**

Introduction

- Being too optimistic about crash guarantees leads to serious **data losses**
- Being too conservative is **expensive** in energy, performance, and hardware lifespan

Background

The POSIX file system interface

- POSIX standard defines a set of system calls for fs access

operation	description
`open`	allocate file descriptor
`write, read`	perform file operations
`link, unlink, mkdir`	perform directory operations
`sync, fsync`	explicitly flush data to disk
`close`	deallocate file descriptor

- `fsync` system call is key to provide data integrity

Crash-Consistency Models

- Used to define permissible states of a file system after a crash
- Consist of:
  - *Litmus tests*: demonstrate allowed/forbidden behaviors of file systems across crashes
  - *Formal specifications*: logic and state machines, describing crash-consistency behavior axiomatic and operational

Crash-Consistency Models

Litmus tests

Litmus tests consist of three parts:

1. Initial setup (optional)

Example:

```
initial:
  f ← creat("f", 0600)
```

Crash-Consistency Models

Litmus tests

Litmus tests consist of three parts:

1. Initial setup (optional)
2. Main body

Example:

```
initial:
  f ← creat("f", 0600)
main:
  write(f, "data")
  fsync(f)
  mark("done")
  close(f)
```

Crash-Consistency Models

Litmus tests

Litmus tests consist of three parts:

1. Initial setup (optional)
2. Main body
3. Final checking

Example:

```
initial:
  f ← creat("f", 0600)
main:
  write(f, "data")
  fsync(f)
  mark("done")
  close(f)
exists?:
  marked("done")
∧ content("f") ≠ "data"
```

Crash-Consistency Models

Litmus tests: Prefix-append (PA)

- The prefix-append (PA) litmus test checks whether, in the event of a crash, a file always contains a prefix of the data that has been appended to it

```
initial:
  N ← 2500
  as, bs ← "a" * N, "b" * N
  f ← creat("file", 0600)
  write(f, as)
main:
  write(f, bs)
exists?:
  content("file") ⊄ as + bs
```

- Also known as "safe-append"
- Popular file systems (`ext4`) do not guarantee this property

Crash-Consistency Models

Litmus tests: Atomic-replace-via-rename (ARVR)

- ARVR checks whether replacing file contents via `rename` is atomic across crashes

```
initial:
  g ← creat("file", 0600)
  write(g, old)
main:
  f ← creat("file.tmp", 0600)
  write(f, new)
  rename("file.tmp", "file")
exists?:
  content("file") ≠ old
∧ content("file") ≠ new
```

Crash-Consistency Models

Formal specifications

Two styles of specification:

- *axiomatic*: describe valid crash behaviors declaratively, using a set of axioms and ordering relations
- *operational*: abstract machines that simulate relevant aspects of file system behavior

Crash-Consistency Models

Formal specifications

Two styles of specification:

- *axiomatic*: describe valid crash behaviors declaratively, using a set of axioms and ordering relations
- *operational*: abstract machines that simulate relevant aspects of file system behavior

Example models will be shown for:

- `seqfs`, an ideal file system with strong crash consistency guarantees
- `ext4`, a real file system with weak consistency guarantees

Crash-Consistency Models

Axiomatic specifications

- Consist of a set of rules/axioms, that specify whether a given execution of a program is allowed

Definition (Program)

- Sequence of atomic events, updating an underlying file system

Axiomatic specifications

File Systems

Definition

- *file system* $\sigma = \langle \sigma_{meta}, \sigma_{data} \rangle$

Axiomatic specifications

File Systems

Definition

- *file system* $\sigma = \langle \sigma_{meta}, \sigma_{data} \rangle$
- σ_{meta}: map from *(object identifiers)* $\mapsto \mathbb{N}$

Axiomatic specifications

File Systems

Definition

- *file system* $\sigma = \langle \sigma_{meta}, \sigma_{data} \rangle$
- σ_{meta}: map from *(object identifiers)* $\mapsto \mathbb{N}$
- σ_{data}: map from $\mathbb{N} \mapsto$ *(file system objects)*

Axiomatic specifications

File Systems

Definition

- *file system* $\sigma = \langle \sigma_{meta}, \sigma_{data} \rangle$
- σ_{meta}: map from *(object identifiers)* $\mapsto \mathbb{N}$
- σ_{data}: map from $\mathbb{N} \mapsto$ *(file system objects)*
- Object identifiers $i \in \mathbb{I}$

Axiomatic specifications

File Systems

Definition

- *file system* $\sigma = \langle \sigma_{meta}, \sigma_{data} \rangle$
- σ_{meta}: map from *(object identifiers)* $\mapsto \mathbb{N}$
- σ_{data}: map from $\mathbb{N} \mapsto$ *(file system objects)*
- Object identifiers $i \in \mathbb{I}$
- Object: $\sigma(i) = \sigma_{data}(\sigma_{meta}(i))$

Axiomatic specifications

File Systems

Definition

- *file system* $\sigma = \langle \sigma_{meta}, \sigma_{data} \rangle$
- σ_{meta}: map from *(object identifiers)* $\mapsto \mathbb{N}$
- σ_{data}: map from $\mathbb{N} \mapsto$ *(file system objects)*
- Object identifiers $i \in \mathbb{I}$
- Object: $\sigma(i) = \sigma_{data}(\sigma_{meta}(i))$
- $\sigma(i) = \perp$ if object $\sigma(i)$ has not been created

Axiomatic specifications

File Systems

Definition

- *file system* $\sigma = \langle \sigma_{meta}, \sigma_{data} \rangle$
- σ_{meta}: map from *(object identifiers)* $\mapsto \mathbb{N}$
- σ_{data}: map from $\mathbb{N} \mapsto$ *(file system objects)*
- Object identifiers $i \in \mathbb{I}$
- Object: $\sigma(i) = \sigma_{data}(\sigma_{meta}(i))$
- $\sigma(i) = \perp$ if object $\sigma(i)$ has not been created
- File object: $\sigma(f) = \langle b, m \rangle$

  b: finite string of bits

  m: finite key-value map of file metadata

Axiomatic specifications

Events

Definition

- Atomic access to a file system σ
- *Update events*: modify objects in σ
- *Synchronization events*: synchronize accesses to (parts of) σ

Axiomatic specifications

Events: Update

Update events include writes to file (meta-) data:

event	operation
$write(f, a, d)$	$b[a] = d$ for $\sigma(f) = \langle b, m \rangle$
$setattr(f, k, v)$	$m[k] = v$ for $\sigma(f) = \langle b, m \rangle$
$extend(f, a, d, s)$	$m[''size''] = s$ and $b[a] = d$
$link(i_1, i_2)$	$\sigma_{meta}(i_2) = \sigma_{meta}(i_1)$
$unlink(i)$	$\sigma_{meta}(i) = \perp$

Axiomatic specifications

Events: Synchronization

Synchronization events include write barriers, non-file-system events (e.g. network-messages) and begin/end of a transaction:

event	operation
$fsync(i)$	sync. accesses to $\sigma(i)$
$sync()$	$fsync(i), \forall i \in \mathbb{I}$
$mark(l)$	mark event identified by unique label l
$begin()$	begins a new transaction
$commit()$	ends the current transaction

Axiomatic specifications

Traces

Definition

- t_P: sequence of file system events of a program P
- $\leq_{t_P}$: total order on events induced by the trace $t_P : e_1 \leq_{t_P} e_2$ iff e_1 occurs before e_2 in the trace t_P
- τ_P: *canonical trace* of P, which is free of crashes and is a strict sequential execution of P
- c_P: *crash* trace - prefix of a *valid* trace t_p that respects transactional semantics:
  c_P contains the same number of *begin* and *commit* events

Axiomatic specifications

Traces

Trace t_P is valid, iff:

- t_P is a permutation of τ_P
- t_P respects sync. semantics of τ_P:
  $e_i \leq_{t_P} e_j$ when $e_i \leq_{\tau_P} e_j$ and any of the following hold:
  - e_i is an *fsync*, *sync*, *mark*, *begin* or *commit* event
  - e_j is a *sync*, *mark*, *begin*, or *commit* event
  - e_j is an *fsync* event on i and e_i is an update event on i
- t_P respects the update semantics of τ_P:
  state of σ after applying t_P = state of σ after applying τ_P

Axiomatic specifications

Example: Ordered-two-file-overwrites

```
initial:
  f ← creat("f", 0600)
  g ← creat("g", 0600)
  write(f, "0")
  write(g, "0")
main:
  pwrite(f, "1", 0)
  pwrite(g, "1", 0)
  fsync(g)
exists?:
  content("f") = "0"
∧ content("g") = "1"
```

- inital state:
  - $\sigma(f) = \sigma(g) = \langle b, m \rangle$
  - $b = "0"$
  - $m = \{permission \mapsto "0600", \dots\}$
- $\tau_P = [e_0, e_1, e_2]$, with:
  - $e_0 = write(f, 0, "1")$
  - $e_1 = write(g, 0, "1")$
  - $e_2 = fsync(g)$
- valid traces:
  - $t_0 = [e_1, e_0, e_2]$
  - $t_1 = [e_1, e_2, e_0]$

Crash-Consistency Models

Definition

- Crash-consistency models determine which valid program traces are permissible
- Strongest model:
  Sequential Crash-Consistency (SCC)
  - Permits no re-ordering of events
  - Only valid trace: τ_P

Crash-Consistency Models

Definition

- Crash-consistency models determine which valid program traces are permissible
- Strongest model:
  Sequential Crash-Consistency (SCC)
  - Permits no re-ordering of events
  - Only valid trace: τ_P
- **But:** Real file systems implement weaker models with additional valid traces

Crash-Consistency Models

Definition

- Crash-consistency models determine which valid program traces are permissible
- Strongest model:
  Sequential Crash-Consistency (SCC)
  - Permits no re-ordering of events
  - Only valid trace: τ_P
- **But:** Real file systems implement weaker models with additional valid traces

Definition

- Let M be a crash-consistency model
- $M\ permits\ t_P \Leftrightarrow M(t_P, \tau_P) = TRUE$

Crash-Consistency Models

Sequential Crash-Consistency (SCC)

Definition

- $SCC(t_P, \tau_P) = TRUE \Leftrightarrow t_P = \tau_P$

Crash-Consistency Models

Sequential Crash-Consistency (SCC)

Definition

- $SCC(t_P, \tau_P) = TRUE \Leftrightarrow t_P = \tau_P$

Example: Ordered-two-file-overwrites

```
initial:
  f ← creat("f", 0600)
  g ← creat("g", 0600)
  write(f, "0")
  write(g, "0")
main:
  pwrite(f, "1", 0)
  pwrite(g, "1", 0)
  fsync(g)
exists?:
    content("f") = "0"
  ∧ content("g") = "1"
```

Crash-Consistency Models

Sequential Crash-Consistency (SCC)

Check whether SCC allows the surprising behavior of this litmus test:

1. Build all valid traces: SCC only permits
   $\tau_P = [write_f, write_g, fsync_g]$
2. Build all crash-traces (prefixes) of the valid traces:

Crash-Consistency Models

Sequential Crash-Consistency (SCC)

Check whether SCC allows the surprising behavior of this litmus test:

1. Build all valid traces: SCC only permits
   $\tau_P = [write_f, write_g, fsync_g]$
2. Build all crash-traces (prefixes) of the valid traces:

	crashtrace	`content(f)`	`content(g)`
c_{P_0}	*crash*	"0"	"0"

Crash-Consistency Models

Sequential Crash-Consistency (SCC)

Check whether SCC allows the surprising behavior of this litmus test:

1. Build all valid traces: SCC only permits
   $\tau_P = [write_f, write_g, fsync_g]$
2. Build all crash-traces (prefixes) of the valid traces:

	crashtrace	`content(f)`	`content(g)`
c_{P_0}	*crash*	"0"	"0"
c_{P_1}	$write_f, crash$	"1"	"0"

Crash-Consistency Models

Sequential Crash-Consistency (SCC)

Check whether SCC allows the surprising behavior of this litmus test:

1. Build all valid traces: SCC only permits $\tau_P = [write_f, write_g, fsync_g]$
2. Build all crash-traces (prefixes) of the valid traces:

	crashtrace	`content(f)`	`content(g)`
c_{P_0}	$crash$	"0"	"0"
c_{P_1}	$write_f, crash$	"1"	"0"
c_{P_2}	$write_f, write_g, crash$	"1"	"1"

Crash-Consistency Models

Sequential Crash-Consistency (SCC)

Check whether SCC allows the surprising behavior of this litmus test:

1. Build all valid traces: SCC only permits
   $\tau_P = [write_f, write_g, fsync_g]$
2. Build all crash-traces (prefixes) of the valid traces:

	crashtrace	`content(f)`	`content(g)`
c_{P_0}	*crash*	"0"	"0"
c_{P_1}	*write$_f$, crash*	"1"	"0"
c_{P_2}	*write$_f$, write$_g$, crash*	"1"	"1"

$\Rightarrow$ SCC does not allow
`content("f") = "0"` $\wedge$ `content("g") = "1"`

Crash-Consistency Models

Definition

Definition (Weaker Crash-Consistency Model)

- Model M_1 is weaker than M_2 iff M_1 permits a superset of the valid traces permitted by M_2
- $M_2(t_P, \tau_P) \Rightarrow M_1(t_P, \tau_P)$

Crash-Consistency Models

`ext4` Crash-Consistency

Definition

- t_P is `ext4` *crash-consistent* iff $e_i \leq_{t_P} e_j$, $\forall e_i, e_j$ such that $e_i \leq_{\tau_P} e_j$ and at least one of the following conditions holds:

  1. e_i and e_j are metadata updates to the same file
  2. e_i and e_j are writes to the same block in the same file
  3. e_i and e_j are updates to the same directory
  4. e_i is a write and e_j is an extend to th same file

Crash-Consistency Models

ext4 Crash-Consistency example: Ordered-two-file-overwrites

Check whether ext4 allows the surprising behavior of this litmus test:

1. Build all valid traces - ext4 permits:
   - $\tau = [write_f, write_g, fsync_g]$
   - $t_0 = [write_g, write_f, fsync_g]$
   - $t_1 = [write_g, fsync_g, write_f]$

Crash-Consistency Models

`ext4` Crash-Consistency example: Ordered-two-file-overwrites

Check whether `ext4` allows the surprising behavior of this litmus test:

1. Build all valid traces - `ext4` permits:
   - $\tau = [write_f, write_g, fsync_g]$
   - $t_0 = [write_g, write_f, fsync_g]$
   - $t_1 = [write_g, fsync_g, write_f]$
2. Build all crash-traces (prefixes) of the valid traces:

	crashtrace	`content(f)`	`content(g)`
c_{P_0}	*crash*	"0"	"0"

Crash-Consistency Models

ext4 Crash-Consistency example: Ordered-two-file-overwrites

Check whether ext4 allows the surprising behavior of this litmus test:

1. Build all valid traces - ext4 permits:
   - $\tau = [write_f, write_g, fsync_g]$
   - $t_0 = [write_g, write_f, fsync_g]$
   - $t_1 = [write_g, fsync_g, write_f]$
2. Build all crash-traces (prefixes) of the valid traces:

	crashtrace	content(f)	content(g)
c_{P_0}	*crash*	"0"	"0"
c_{P_1}	$write_g$, *crash*	"0"	"1"

Crash-Consistency Models

`ext4` Crash-Consistency example: Ordered-two-file-overwrites

Check whether `ext4` allows the surprising behavior of this litmus test:

1. Build all valid traces - `ext4` permits:
   - $\tau = [write_f, write_g, fsync_g]$
   - $t_0 = [write_g, write_f, fsync_g]$
   - $t_1 = [write_g, fsync_g, write_f]$
2. Build all crash-traces (prefixes) of the valid traces:

	crashtrace	`content(f)`	`content(g)`
cp_0	*crash*	"0"	"0"
cp_1	$write_g, crash$	"0"	"1"

$\Rightarrow$ `ext4` allows

`content("f") = "0"` $\wedge$ `content("g") = "1"`

Crash-Consistency Models

Operational specifications

- Non-deterministic state machine M
- M is modeled as $\langle \sigma, p \rangle$, with σ as the file system and p as a program counter for P

Crash-Consistency Models

Operational specifications: Examples

$$\frac{\sigma' = \texttt{FLUSH}(\texttt{APPLY}(P[p], \sigma))}{\langle \sigma, p \rangle \mapsto \langle \sigma', p+1 \rangle}\texttt{STEPSEQ}$$

$$\frac{}{\langle \sigma, p \rangle \mapsto \langle \sigma, \perp \rangle}\texttt{CRASH}$$

Figure: An operational model for SCC

Bornholt *et al.*, "Specifying and Checking File System Crash-Consistency Models"

Crash-Consistency Models

Operational specifications: Examples

$$\frac{\sigma' = \texttt{FLUSH}(\texttt{APPLY}(P[p], \sigma))}{\langle \sigma, p \rangle \mapsto \langle \sigma', p+1 \rangle}\texttt{STEPSEQ}$$

$$\frac{}{\langle \sigma, p \rangle \mapsto \langle \sigma, \perp \rangle}\texttt{CRASH}$$

Figure: An operational model for SCC

$$\frac{\sigma' = \texttt{APPLY}(P[p], \sigma)}{\langle \sigma, p \rangle \mapsto \langle \sigma', p+1 \rangle}\texttt{STEP}$$

$$\frac{}{\langle \sigma, p \rangle \mapsto \langle \sigma, \perp \rangle}\texttt{CRASH}$$

$$\frac{\sigma' = \texttt{PARTIALFLUSH}(\sigma)}{\langle \sigma, p \rangle \mapsto \langle \sigma', p \rangle}\texttt{NONDET}$$

Figure: An operational model for `ext4`

Bornholt *et al.*, "Specifying and Checking File System Crash-Consistency Models"

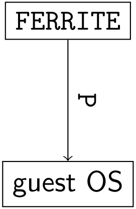
FERRITE

- Suite for reasoning about crash-consistency models
- Consists of two tools to explore all possible crash behaviors of a litmus test:
  - **Explicit enumerator**: executes litmus tests against actual file system implementations
  - **Bounded model checker**: executes litmus tests symbolically against an axiomatic specification

FERRITE

Explicit Enumerator: Executing tests against file system implementations

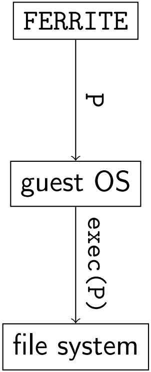
1. Enumerator forwards litmus test on guest OS



FERRITE

Explicit Enumerator: Executing tests against file system implementations

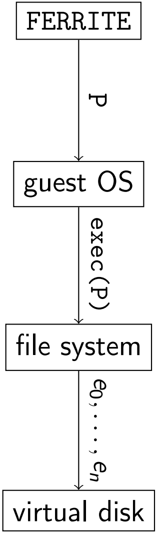
1. Enumerator forwards litmus test on guest OS
2. Guest OS executes litmus test



FERRITE

Explicit Enumerator: Executing tests against file system implementations

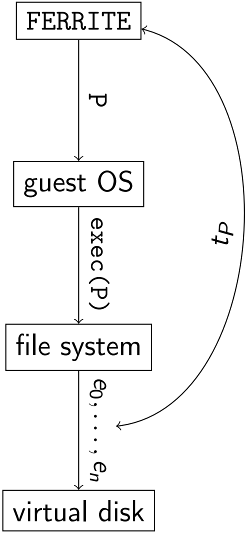
1. Enumerator forwards litmus test on guest OS
2. Guest OS executes litmus test
3. File system issues events to the virtual disk



FERRITE

Explicit Enumerator: Executing tests against file system implementations

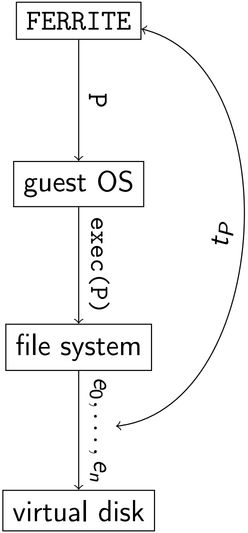
1. Enumerator forwards litmus test on guest OS
2. Guest OS executes litmus test
3. File system issues events to the virtual disk
4. FERRITE records events in a trace



FERRITE

Explicit Enumerator: Executing tests against file system implementations

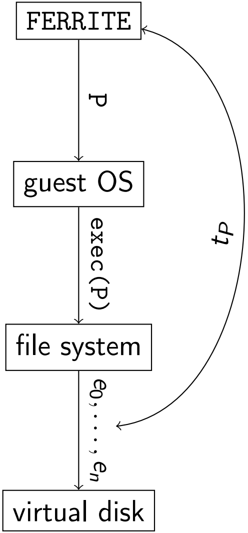
1. Enumerator forwards litmus test on guest OS
2. Guest OS executes litmus test
3. File system issues events to the virtual disk
4. `FERRITE` records events in a trace
5. Enumerator prod. all possible reorderings and prefixes



FERRITE

Explicit Enumerator: Executing tests against file system implementations

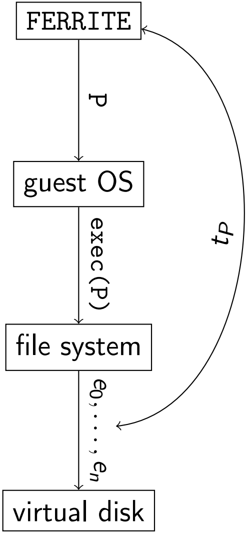
1. Enumerator forwards litmus test on guest OS
2. Guest OS executes litmus test
3. File system issues events to the virtual disk
4. `FERRITE` records events in a trace
5. Enumerator prod. all possible reorderings and prefixes
6. Each prefix prod. a disk image file corresponding to a disk state after a crash



FERRITE

Explicit Enumerator: Executing tests against file system implementations

1. Enumerator forwards litmus test on guest OS
2. Guest OS executes litmus test
3. File system issues events to the virtual disk
4. `FERRITE` records events in a trace
5. Enumerator prod. all possible reorderings and prefixes
6. Each prefix prod. a disk image file corresponding to a disk state after a crash
7. Enumerator mounts disk image and checks predicates in the given litmus test



FERRITE

Bounded model checker: Executing tests against specifications

- Input:
  - litmus test P
  - axiomatic specification of a crash-consistency model M
- Checker prod. set of crash traces
  $T = \{c_P | c_P \text{ is a crash trace of } t_P \text{ and } M(t_P, \tau_P)\}$
- $\forall c_P \in T$: checks whether c_P satisfies the predicates in P

Conclusion

The problem:

- POSIX standards under-specify guarantees that file systems should provide in the face of crashes

Conclusion

The problem:

- `POSIX` standards under-specify guarantees that file systems should provide in the face of crashes
- Application writers can only guess how to achieve data integrity *and* improve performance

Conclusion

The problem:

- `POSIX` standards under-specify guarantees that file systems should provide in the face of crashes
- Application writers can only guess how to achieve data integrity *and* improve performance

The solution:

- *Crash-consistency models* (i.e. litmus test and formal specifications) build a contract between the file system and the application

Outlook

Synthesis: Proof-of-concept

- Crash-consistency specifications allow to *synthesize* desired crash-safety properties

Outlook

Synthesis: Proof-of-concept

- Crash-consistency specifications allow to *synthesize* desired crash-safety properties
- Application writer develops P assuming SCC

Outlook

Synthesis: Proof-of-concept

- Crash-consistency specifications allow to *synthesize* desired crash-safety properties
- Application writer develops P assuming SCC
- Synthesizer transforms P, by inserting a minimal set of barriers (i.e. `fsync`)

Outlook

Synthesis: Proof-of-concept

- Crash-consistency specifications allow to *synthesize* desired crash-safety properties
- Application writer develops P assuming SCC
- Synthesizer transforms P, by inserting a minimal set of barriers (i.e. `fsync`)
- Resulting program P' behaves just like P under a given weaker crash-consistency model (e.g. `ext4`)