Spezifizieren und Überprüfen von Dateisystem Absturz-Konsistenz Modellen

Steven Lang
Johannes Gutenberg Universität Mainz
slang03@students.uni-mainz.de

Abstract

Die heutigen POSIX Dateisystem Schnittstellen besitzen keine klare Definition der resultierenden Zustände eines Systemabsturzes. Dies ist problematisch für Anwendungen, welche auf persistenten Speicher zurückgreifen um den ursprünglichen Zustand nach einem solchen Systemabsturz wiederherzustellen. Die Schwierigkeit für Entwickler besteht nun darin, die Reihenfolge und Abhängigkeit zwischen Dateisystem Operationen zu verstehen. Dies kann zu unvorhersehbarem Fehlverhalten, korrupten Anwendungszuständen und im schlimmsten Fall sogar zum Verlust der Daten führen.

Diese Ausarbeitung beschreibt Absturz-Konsistenz Modelle, analog zu Speicher-Konsistenz Modellen. Sie bestehen aus den folgenden zwei Bestandteilen: Litmus Tests, welche erlaubte und verbotene Verhalten von Dateisystemen zeigen und axiomatische Spezifikationen, welche zulässige Absturzverhalten deklarativ mit einer Menge an Axiomen beschreiben, sowie operationale Spezifikationen, bestehend aus abstrakten Automaten, die relevante Aspekte eines Dateisystem simulieren können.

Des weiteren wird das Framework `FERRITE` beschrieben, mit dem man Absturz-Konsistenz Modelle ausarbeiten und gegen echte Dateisysteme validieren kann.

1. Motivation

Persistenter Speicher ist heutzutage Hauptbestandteil vieler Applikationen und wird verwendet um Daten, über die Lebensdauer des Applikationsprozesses hinweg, aufrecht zu erhalten. POSIX bietet hierfür generalisierte Schnittstellen, an denen sich Applikationsentwickler bedienen können. Um tatsächliche Integrität der Daten zu erhalten müssen diese Schnittstellen richtig verstanden und verwendet werden.

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

Abbildung 1: Replace-via-rename Pattern - Dieses Schema soll atomarität der Ersetzen-Operation simulieren

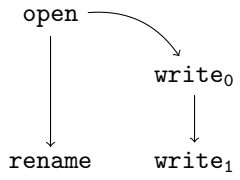

Abbildung 2: `ext4` Garantien über Einhaltungen der Reihenfolgen zwischen verschiedenen Operationen

Durch falsche Verwendung kann es zu umgehender Korruption und im schlimmsten Fall zum Verlust der Daten kommen. Dass das Verstehen solcher Schnittstellen kein trivialer Prozess ist zeigt eine Studie von Pillai et al. [3] und Zheng et al. [5]. Hierbei geht hervor, dass selbst vollentwickelte Anwendungen wie LevelDB, SQLite und Git falsche Annahmen über das Verhalten von Dateisystemen machen.

Ein bekanntes Beispiel für solche falschen Annahmen ist das sog. replace-via-rename Schema in Abbildung 1. Hierbei wird eine Datei ‘‘`file`’’ ersetzt indem eine temporäre Datei ‘‘`file.tmp`’’ geöffnet, beschrieben und schließlich zu ‘‘`file`’’ umbenannt wird. Dieses Schema hat das Ziel eine Datei atomar zu aktualisieren. Abbildung 2 zeigt die durch `ext4` garantierten Ausführungsreihenfolgen zwischen den einzelnen Operationen `open`, `write`$_i$ (Zerlegung der ursprünglichen `write`-Operation in kleinere Blöcke) und `rename`. So ist zum einen gegeben, dass die Datei zuerst beschrieben werden kann nachdem sie geöffnet wurde. Zum anderen wird auch garantiert, dass die Datei zuerst geöffnet sein muss bevor sie umbenannt wird. Worüber `ext4` jedoch keine Aussage trifft ist die Reihenfolge zwischen dem

Datei-Festplattenzustand	Mögliche Ausführungen auf der Festplatte
neu	`open, write`$_0$`, write`$_1$`, rename, ...`
	`open, write`$_0$`, rename, write`$_1$`, ...`
	`open, rename, write`$_0$`, write`$_1$`, ...`
alt	`open, crash`
	`open, write`$_0$`, crash`
	`open, write`$_0$`, write`$_1$`, crash`
leer	`open, rename, crash`
teilweise neu	`open, write`$_0$`, rename, crash`
	`open, rename, write`$_0$`, crash`

Tabelle 1: Unerwartete Dateizustände nach verschiedenen Ausführungsreihenfolgen

Beschreiben und dem Umbenennen der Datei. Dies wirkt sich einerseits zwar positiv auf die Performanz aus, führt jedoch im Falle eines Systemabsturz zu unerwünschten Dateizuständen, welche in Tabelle 1 zu sehen sind. Es kann passieren, dass `rename` nach dem ersten und vor dem zweiten `write` eintrifft, das System abstürzt und resultierend eine Datei in einem Zustand hinterlässt, der weder den alten Daten, noch den neuen Daten entspricht. Dies widerspricht somit dem Zweck des Ersetzen-durch-Umbenennung Schemas, welches eine atomare Aktualisierung bieten soll. Eine Möglichkeit diesen Zustand zu verhindern wäre ein zusätzliches `fsync(fd)` durchzuführen, bevor die Datei mit `close(fd)` geschlossen wird. Die Datei ‘‘`file`’’ wird folglich immer entweder die alten oder neuen Daten beinhalten.

2. Einführung

Moderne Dateisystem-Optimierungen erlauben es die Reihenfolge in der Operationen ausgeführt werden zu relaxieren. Diese Relaxation kann erhebliche Performanzgewinne einbringen, ist jedoch unsichtbar auf Applikationsebene. Applikationsentwickler nehmen ein bestimmtes Verhalten und Garantien über die Konsistenz eines Dateisystems nach einem Systemabsturz an und entwickeln ihre Applikationen unter diesen Annahmen. Optimismus hinsichtlich dieser Garantien kann zu Korruption, sowie Verlust der Daten führen. Pessimismus über Konsistenz-Garantien ist jedoch teuer bezüglich Energie, Performanz und Hardwarelebensdauer. Es ist deshalb wichtig eine genaue Beschreibung über das Verhalten eines Dateisystem bei Systemabstürzen zu liefern. Das Hauptproblem für Anwendungsentwickler besteht somit darin, dieses Verhalten zu verstehen.

Bornholt et al. [2] beschreiben als Lösung sogenannte Absturz-Konsistenz Modelle für Dateisysteme – analog zu Speicher-Konsistenz Modellen [1, 4]. Sie skizzieren Relaxationen axiomatisch (§5.1) und operational (§5.4). Diese Modelle bestehen aus zwei Hauptbestandteilen:

- *Litmus Tests*: Kleine Programme die erlaubte und unerlaubte Verhalten eines Dateisystems nach einem Systemabsturz zeigen.

Operation	Beschreibung
`open`	alloziert Datei-Deskriptor
`write, read`	auführen von Datei-Operationen
`link, unlink, mkdir`	auführen von Ordner-Operationen
`sync, fsync`	explizites schreiben der Daten auf den Hintergrundspeicher
`close`	gibt Datei-Deskriptor frei

Tabelle 2: POSIX Standard Systemaufrufe für Dateisystemzugriffe

- *Formale Spezifikationen*: Logik und (nichtdeterministische) Automaten als axiomatische und operationale Beschreibung von Absturz-Konsistenz Modellen

So würde das Schema in Abbildung 1 *sequentielle Absturz-Konsistenz* fordern um die ursprüngliche Reihenfolge beizubehalten und die Integrität der Datei zu garantieren. Im Gegensatz zur sequentiellen Absturz-Konsistenz erlaubt das Modell eines `ext4`-Dateisystems das vertauschen einer Schreib- und Umbenennungsoperation, welches in den in Abbildung 1 beschriebenen Dateizuständen resultiert.

Des weiteren wird das Framework `FERRITE` vorgestellt, mit dem man Litmus Tests zum einen gegen formale Spezifikationen eines Absturz-Konsistenz Modells, zum anderen aber auch gegen echte Dateisysteme testen kann. Dies dient zur Überprüfung der Übereinstimmung zwischen Spezifikation und Modell, sowie der allgemeinen Validierung des Modells an echten Dateisystemen.

Im fortführenden Teil dieser Ausarbeitung wird in §3 auf nötiges Hintergrundwissen eingegangen. §4 beschreibt die sog. Litmus Tests. §5 erklärt formale Spezifikationen von Absturz-Konsistenz Modellen. §6 zeigt die Funktionen des Frameworks FERRITE. §8 liefert ein knappes Fazit. §7 gibt einen Ausblick auf mögliche Anwendungen.

3. Hintergrund

Um eine einheitliche und universelle Schnittstelle zwischen Anwendungen und Betriebssystemen zu bilden, definiert POSIX einen Standard an Systemaufrufen für den Dateisystemzugriff.

3.1 Die POSIX Dateisystem-Schnittstelle

In Tabelle 2 sind die wichtigsten Operationen der POSIX Dateisystem-Schnittstelle aufgelistet. So können Anwendungen beginnend mit `open` einen Datei-Deskriptor belegen um im Anschluss Datei-Operationen wie `read` und `write` auf der geöffneten Datei auszuführen. Das Schreiben einer Datei in den Hintergrundspeicher wird mittel `fsync` (bzw. `sync` für alle Dateien) sichergestellt. Im Anschluss kann mittels `close` der Datei-Deskriptor, sowie Speicherplatz im Hauptspeicher freigegeben werden. Da jedoch die eigentliche Ausführung der Datei- und Ordneroperationen

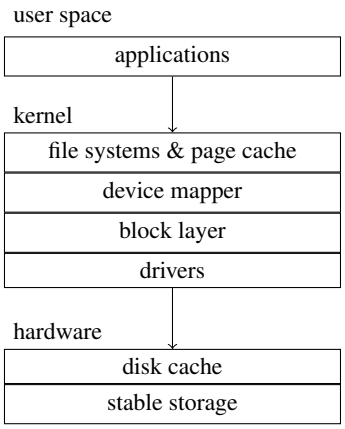

Abbildung 3: Linux I/O Pfad Beispiel - Von der Anwendung zur Hardware

im Hauptspeicher zwischengespeichert werden kann, ist der Aufruf von `fsync` zur Synchronisation aller geöffneten Datei-Deskriptoren mit dem Hintergrundspeicher die Schlüsselkomponente um Datenintegrität zu gewährleisten. Die POSIX Definition von `fsync` garantiert explizit bei einem Systemabsturz den Zustand des Dateisystem bis zu dem Zeitpunkt bevor `fsync` aufgerufen wurde. Darüber hinaus gibt POSIX jedoch keine weiteren Garantien, in welchem Zustand sich das Dateisystem nach einem Absturz befindet, während einer der anderen Operationen in Tabelle 2 ausgeführt wird.

3.2 Relaxation der Reihenfolge

Moderne Betriebssysteme erlauben es die Reihenfolge, in der Dateisystem-Operationen im Anwendungsbereich aufgerufen werden, zu verändern um diese effizienter zu gruppieren und auszuführen. Diese Veränderungen der Reihenfolge kann individuell von jeder Schicht im I/O Pfad des Betriebssystems (Beispiel Linux Abbildung 3) vorgenommen werden. Da die Anwendungen aus einem Zwischenspeicher lesen, sind die Relaxationen vorerst nicht sichtbar. Kommt es während der Ausführung jedoch zu einem Systemabsturz, kann dies unerwartete Zustände hervorbringen. Bornholt et al. [2] haben sich somit vorgenommen, diesen Unklarheiten mittels einer präzisen Beschreibung der Dateisystem-Absturzgarantien entgegenzuwirken ohne die Notwendigkeit tiefer Kenntnisse über die I/O Schichten des darüberliegenden Betriebssystems zu besitzen.

4. Litmus Tests

Litmus Tests sind kleine Programme, die ein Dateisystem auf bestimmte Zustände nach der Ausführung von Operationen überprüfen. Die Überprüfung soll gewisse Garantien herausfinden, welche das zu prüfende Dateisystem liefert. So kann zum Beispiel überprüft werden, ob das Ausführen eines Schreibbefehls die Ausführung eines vorhergehenden Schreibbefehls impliziert. Somit kann man zuverlässig Dateisystem Absturz-Garantien mithilfe von Litmus Tests inspizieren und offenlegen.

4.1 Aufbau

Ein Litmus Test besitzt drei Hauptbestandteile:

- **Initialer Aufbau**: Der Ausgangspunkt für den Litmus Test ist ein leeres Dateisystem. Der initiale Aufbau soll Bedingungen herstellen, unter denen der Test ausgeführt wird. So können hier zum Beispiel Dateien angelegt, beschrieben oder verlinkt werden. Dieser Teil des Tests wird immer mit einem impliziten `sync` abgeschlossen, sodass alle Veränderungen in den Hintergrundspeicher geschrieben werden. Der initiale Aufbau ist optional und wird mit ‘‘`initial:`’’ gekennzeichnet.
- **Körper**: Der Körper des Litmus Tests wird nach dem initialen Aufbau ausgeführt. Es wird eine Pseudofunktion `mark` eingeführt, die einen bestimmten Abschnitt mit einer Kennung markiert, sobald `mark` ausgeführt wird. Das Programm kann zu jedem Zeitpunkt in diesem Teil des Tests abstürzen. Der Körper des Litmus Tests wird mit ‘‘`main`’’ gekennzeichnet.
- **Finales Prüfen**: Im letzten Teil des Litmus Tests können bestimmte Zustände überprüft werden. So kann mit dem Prädikat `marked(`*`label`*`)` abgefragt werden, ob ein bestimmter Abschnitt mittels `mark` bereits erreicht wurde oder mit dem Prädikat `content(`*`name`*`)` der Inhalt einer Datei mit einem eigens definierten String abgeglichen werden. Die Prädikate werden mit logischen Operatoren verknüpft. Diese Überprüfung wird mit ‘‘`exists?:`’’ eingeleitet.

  Das folgende Beispiel zeigt einen Litmus Test:

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

Hierbei wird im initialen Aufbau eine leere Datei `f` angelegt. Im Körper des Litmus Tests wird `f` mit dem String ‘‘`data`’’ beschrieben und synchronisiert. Anschließend wird die Kennung ‘‘`done`’’ markiert und die Datei geschlossen. Im letzten Teil des Litmus Tests wird nun überprüft, ob es einen Zustand geben kann, in dem die Kennung ‘‘`done`’’ bereits markiert wurde, der Inhalt der Datei `f` jedoch nicht dem String ‘‘`data`’’ entspricht.

4.2 Anwendung

Im Folgenden werden Litmus Tests verwendet um unintuitive und überraschende Verhalten von Dateisystem aufzuzeigen.

Prefix-append (PA). Dieser Litmus Test überprüft ob beim Anhängen von Daten an eine bereits vorhandene Datei das Resultat stets ein Präfix der Daten ist, die man bei erfolgreichem Ausführen der Operationen erwarten würde.

```
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
```

Im initialen Aufbau wird eine Datei `f` angelegt und mit einer Kette von aufeinanderfolgenden `a`'s beschrieben. Im Körper des Litmus Tests wird eine weitere Kette von `b`'s an `f` angehängt. Anschließend wird überprüft, ob eine Ausführungsreihenfolge existiert, in der das Resultat in der Datei `f` kein Präfix der konkatenierten Zeichenketten aus `a`'s und `b`'s ist.

Das Erfüllen dieser Eigenschaft garantiert somit, dass ein Absturz des Systems keine Daten schreibt, die nicht Teil des zu schreibenden Inhalts ‘‘`bs`’’ sind.

Atomic-replace-via-rename (ARVR). Das ARVR Schema überprüft ob das Ersetzen von Dateiinhalten durch die Umbenennung einer temporären Datei eine atomare Operation gegenüber Systemabstürzen darstellt.

```
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
```

Hierbei wird eine Datei ‘‘`file`’’ mit den alten Daten ‘‘`old`’’ initialisiert. Im Anschluss wird eine temporäre Datei geöffnet, mit den neuen Daten ‘‘`new`’’ beschrieben und zu ‘‘`file`’’ umbenannt. Abschließend überprüft man ob es einen möglichen Zustand gibt, in dem die Datei ‘‘`file`’’ weder die alten Daten, noch die neuen Daten beinhaltet.

5. Formale Spezifikationen

Ein Absturz-Konsistenz Model definiert erlaubte Zustände eine Dateisystems nach einem Systemabsturz. Diese können auf zwei Arten und Weisen beschrieben werden:

- **Axiomatisch**: Deklarative Beschreibung valider Absturzverhalten mithilfe von Axiomen und Ordnungsverhältnissen
- **Operational**: Abstrakte (Zustands-) Maschinen, die Dateisystemverhalten zu verschiedenen Zeitpunkten simulieren können

Im Abschnitt XXX werden beide Spezifikationen anhand von zwei verschiedenen Dateisystemen gezeigt. Zum einen wird das `seqf`-Dateisystem untersucht, welches Operationen sequentiell ausführt und keine Relaxationen erlaubt. Zum anderen werden Spezifikationen für das `ext4`-Dateisystem gezeigt, welches in vielen Betriebssystemen zum Einsatz kommt.

Ereignis	Operation
$write(f, a, d)$	$b[a] = d$ mit $\sigma(f) = \langle b, m\rangle$
$setattr(f, k, v)$	$m[k] = v$ mit $\sigma(f) = \langle b, m\rangle$
$extend(f, a, d, s)$	$m[\text{“}size\text{”}] = s$ und $b[a] = d$
$link(i_1, i_2)$	$\sigma_{meta}(i_2) = \sigma_{meta}(i_1)$
$unlink(i)$	$\sigma_{meta}(i) = \perp$

Tabelle 3: Formale Beschreibung der Aktualisierungsereignisse

5.1 Axiomatische Spezifikationen

Axiomatische Spezifikationen bestehen aus Regeln und Axiomen welche eine Ausführungsreihenfolge eines Programms auf Zulässigkeit überprüft. Hierfür werden vorab einige Begriffe definiert:

Dateisysteme

- *Programm*: Eine Sequenz von atomaren Ereignissen e, welche ein zugrundeliegendes Dateisystem σ aktualisieren.
- *Dateisystem*: $\sigma = \langle \sigma_{meta}, \sigma_{data}\rangle$, Tupel bestehend aus Daten und Metadaten
- σ_{meta}: Zuordnung von *(Objektidentifikatoren)* $\mapsto \mathbb{N}$
- σ_{data}: Zuordnung von $\mathbb{N} \mapsto$ *(Dateisystem Objekte)*
- *Objektidentifikatoren:* $i \in \mathbb{I}$
- *Objekt:* $\sigma(i) = \sigma_{data}(\sigma_{meta}(i))$
- $\sigma(i) = \perp$ falls das Objekt $\sigma(i)$ noch nicht angelegt wurde
- *Datei Objekt*: $\sigma(f) = \langle b, m\rangle$, Tupel bestehend aus einer endlichen Kette von Bits b und einer endlichen Schlüssel-Wert Zuweisung an Datei-Metadaten

Ereignisse Ereignisse sind atomare Zugriffe auf ein Dateisystem σ. Diese können unterteilt werden in sog. Aktualisierungsereignisse (Tabelle 3), welche Objekte in σ modifizieren, und Synchronisierungsereignisse (Tabelle 4), die den Zugriff auf das Dateisystem synchron halten.

Spuren Eine *Spur* t_P ist die Sequenz von Dateisystem-Ereignissen eines Programms P. Spuren können eine Ordnung auf Ereignissen erzeugen. So ist die Ordnung $\leq_{t_P}$ wie folgt definiert: $e_1 \leq_{t_P} e_2$ gdw. e_1 ereignet sich vor e_2 in der Spur t_P. Das Programm P besitzt eine kanonische Spur τ_P, welche ohne Abstürze erfolgt und einer strikten sequentiellen Ausführung des Programms P entspricht. Des weiteren wird eine Absturz-Spur c_P als Präfix einer *validen* Spur t_P definiert, welche die transaktionale Semantik von t_P berück-

Ereignis	Operation
$fsync(i)$	Sync. Zugriff auf $\sigma(i)$
$sync()$	$fsync(i), \forall i \in \mathbb{I}$
$mark(l)$	Markiere Ereignis mit einer einzigartigen Kennung l
$begin()$	Startet eine neue Transaktion
$commit()$	Beendet die aktuelle Transaktion

Tabelle 4: Formale Beschreibung der Synchronisierungsereignisse

sichtigt. Das heißt c_P beinhaltet die gleiche Anzahl an *begin* und *commit* Ereignissen wie t_P. Man spricht von einer *validen* Spur, falls sie die folgenden Bedingungen erfüllt:

- t_P ist eine Permutation der kanonischen Spur τ_P
- t_P berücksichtigt die Synchronisations-Semantik von τ_P, d.h $e_i \leq_{t_P} e_j$ wenn $e_i \leq_{\tau_P} e_j$ und eine der folgenden Bedingung erfüllt ist:
  - e_i ist ein $fsync, sync, mark, begin$ oder $commit$ Ereignis
  - e_j ist ein $sync, mark, begin$, or $commit$ Ereignis
  - e_j ist ein $fsync$ Ereignis bzgl. i und e_i ist ein Aktualisierungsereignis bzgl. i
- t_P berücksichtigt die Aktualisierungs-Semantik von τ_P: Der Zustand von σ nach der Anwendung von t_P muss dem Zustand von σ nach der Anwendung von τ_P entsprechen

5.2 Anwendung

Im Folgenden wird die soeben beschriebene Modellierung an einem Beispiel (Ordered-two-file-writes) dargestellt:

```
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"
```

Der initiale Zustand des Dateisystems besteht aus zwei Dateisystem Objekten $\sigma(f) = \sigma(g) = \langle b, m \rangle$ mit $b =$ "0" und $m = \{permission \mapsto$ "0600", $\dots\}$. Es ergibt die kanonische Spur $\tau_P = [e_0, e_1, e_2]$ mit $e_0 = write(f, 0,$ "1"$)$, $e_1 = write(g, 0,$ "1"$)$ und $e_2 = fsync(g)$. Valide Spuren sind nach Definition $t_0 = [e_1, e_0, e_2]$, sowie $t_1 = [e_1, e_2, e_0]$.

	Absturz-Spur	`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"

Tabelle 5: Mögliche Absturz-Spuren der sequentiellen Absturz-Konsistenz bezüglich des Litmus Tests in 5.2

5.3 Absturz-Konsistenz Modelle

Absturz-Konsistenz Modelle bestimmen zulässige Spuren eines Programms *P*. Das stärkste Absturz-Konsistenz Modell ist die sog. *sequentielle Absturz-Konsistenz*. Sie erlaubt keine Umordnung der Reihenfolge der Ereignisse in *P* und besitzt somit nur die kanonische Spur τ_P als valide Spur. Da eine sequentielle Ausführung nicht immer notwendig sein muss implementieren echte Dateisystem meist schwächere Modelle mit weiteren validen Spuren. Ein Absturz-Konsistenz Modell *M* ist wie folgt definiert: *M* erlaubt die Spur t_P, gdw. $M(t_P, \tau_P)$ zu wahr ausgewertet wird. Somit kann man die sequentielle Absturz-Konsistenz definieren als $SCC(t_P, \tau_P) = TRUE \Leftrightarrow t_P = \tau_P$.

Um nun zu überprüfen ob ein Absturz-Konsistenz Modell das Verhalten eines Litmus Tests erlaubt werden alle valide Spuren des Tests gesammelt. Im Falle des *SCC* Modells bezüglich des in 5.2 genannten Litmus Tests wäre dies nur die kanonische Spur $\tau_P = [write_f, write_g, fsync_g]$. Im nächsten Schritt werden alle Absturz-Spuren der validen Spuren gebaut (Tabelle 5) und deren Dateisystem-Zustände mit den Prädikaten des Litmus Tests abgeglichen. Man kann somit erkennen, dass die sequentielle Absturz-Konsistenz nicht das Verhalten des Litmus Tests erlaubt. Da *SCC* jedoch ein starkes Modell ist muss man ebenso schwächere Modelle in Betracht ziehen.

Schwächere Absturz-Konsistenz Modelle Ein Modell M_1 ist schwächer als ein Modell M_2, gdw. M_1 eine Obermenge der validen Spuren zulässt, die auch M_2 erlaubt: $M_2(t_P, \tau_P) \Rightarrow M_1(t_P, \tau_P)$.

ext4 Absturz-Konsistenz Eine Spur t_P ist `ext4` absturzkonsistent, gdw. gilt $e_i \leq_{t_P} e_j$, $\forall e_i, e_j$ sodass $e_i \leq_{\tau_P} e_j$ und mindestens eine der folgenden Bedingungen erfüllt ist:

1. e_i und e_j sind Metadaten-Aktualisierungen der selben Datei
2. e_i und e_j sind Schreibbefehle in den selben Block der selben Datei
3. e_i und e_j sind Aktualisierungen des selben Ordners
4. e_i ist ein Schreib- und e_j ist ein Erweiterungsbefehl auf der selben Datei

Mit der Definition der `ext4` Absturz-Konsistenz ist es nun möglich das Verhalten eines `ext4` Dateisystem bezüg-

	Absturz-Spur	`content(f)`	`content(g)`
c_{P_0}	$crash$	"0"	"0"
$\mathbf{c_{P_1}}$	$\mathbf{write_g, crash}$	"**0**"	"**1**"

Tabelle 6: Mögliche Absturz-Spuren der `ext4` Absturz-Konsistenz bezüglich des Litmus Tests in 5.2

lich des Litmus Tests in 5.2 zu überprüfen. Valide Spuren sind in diesem Fall:

- $\tau = [write_f, write_g, fsync_g]$
- $t_0 = [write_g, write_f, fsync_g]$
- $t_1 = [write_g, fsync_g, write_f]$

Im Anschluss werden sukzessiv alle Absturz-Spuren der validen Spuren gebaut und gegen die Prädikate des Litmus Tests geprüft. Wie man in Tabelle 6 sehen kann, erlaubt die Absturz-Spur c_{P_1} das überraschende Verhalten des Litmus Tests.

5.4 Operationale Spezifikationen

Statt Absturz-Spuren gegen Litmus Tests zu überprüfen, modelliert man in den operationalen Spezifikationen ein Dateisystem mithilfe von nicht-deterministischen Automaten. Ein Dateisystem σ besitzt einen Zustand und besteht somit aus σ_{inCore}, dem volatilen Teil, welcher noch nicht in den persistenten Speicher geschrieben wurde, und σ_{onDisk} als Teil des bereits persistierten Speichers. Beide Speicher sind wiederum Tupel der Form $\langle \sigma_{meta}, \sigma_{data} \rangle$. Ereignisse beeinflussen den volatilen Speicher σ_{inCore}. Der Automat kann sich nicht-deterministisch dazu entscheiden Änderungen in σ_{inCore} mit σ_{onDisk} synchronisieren oder andernfalls abzustürzen. Er wird als Tupel $M = \langle \sigma, p \rangle$ modelliert, wobei *p* den Programmzähler des Programms *P* darstellt.

In Abbildung 4 wird das operationale Modell der sequentiellen Absturz-Konsistenz skizziert. Hierbei repräsentiert die Operation `STEPSEQ` einen sequentiellen Schritt bei der Ausführung des Programms. Es wird die *p*-te Stelle von *P* auf dem Dateisystem σ angewandt und sofort in σ_{onDisk} mithilfe von `FLUSH` persistiert. Das Modell *M* wird von σ zu σ' überführt und der Programmzähler *p* wird um eins inkrementiert. Somit ist in jedem Zustand von *M* die Gleichheit $\sigma_{inCore} = \sigma_{onDisk}$ garantiert. Der Automat kann ebenso nicht-deterministisch abstürzen (Operation `CRASH`) und den Programmzähler auf $\perp$ setzen.

Im operationalen Modell für `ext4` (Abbildung 5) geht dagegen die vorab beschriebene Relaxation einher und verlangt kein direktes persistieren der Daten nach dem Anwenden des *p*-ten Schritts von *P* in der Operation `STEP`. Das Modell ist um eine Operation `NONDET` erweitert, welche es erlaubt Teile von σ_{inCore} mit σ_{onDisk} nicht-deterministisch zu synchronisieren. Hierbei verändert sich der Zustand des

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

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

Abbildung 4: Das operationale Modell für *SCC*

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

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

Abbildung 5: Das operationale Modell für `ext4`

Dateisystems zu σ', jedoch wird der Programmzähler nicht inkrementiert.

6. FERRITE

FERRITE ist ein Framework um Absturz-Konsistenz Modelle gegen echte Dateisysteme zu testen. Es besteht aus zwei Hilfsprogrammen, zum einen den *expliziten Enumerator*, welcher Litmus Tests gegen Dateisystemimplementierungen ausführen kann, zum anderen den *bounded model checker*, der Litmus Tests symbolisch gegen axiomatische Spezifikationen ausführt.

6.1 Expliziter Enumerator

Die Eingabe des Enumerators besteht aus einem Litmus Test *P* und einem gegebenen Zieldateisystem auf dem *P* ausgeführt werden soll. Die Ausgabe soll aus allen möglichen Ausgängen von *P* bestehen. Hierbei entsteht folgendes Problem: Die Systemaufrufe, welche von *P* produziert werden wandern durch mehrere Schichten des Betriebssystem I/O Pfads (Beispiel siehe Abbildung 3). Jede Schicht besitzt ihre eigenen Regeln wie sie diese Systemaufrufe zwischenspeichern und umsortieren kann. Letztendlich kommen an der Hardware vom Betriebssysteme produzierte Disk-Befehle an, welche das Speichermedium wiederum nach eigenen Regeln umsortieren und zwischenspeichern kann. Somit ist es ungemein komplex jede mögliche Ausführungsreihenfolge nachzuvollziehen und zu simulieren. Um zu garantieren, dass alle möglichen Absturzzustände beobachtet werden, abstrahiert FERRITE die an der Hardware ankommenden Befehle in einem sogenannten Disk Modell. Das Standard Disk Modell in FERRITE erlaubt folgende Operationen:

- $write(blockID, data)$: Schreibt $data$ in einen gegebenen Block

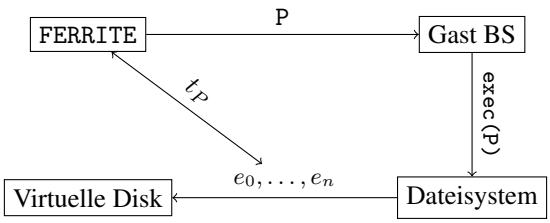

Abbildung 6: Ablauf des expliziten Enumerators

- $flush()$: Persistiert Zwischenspeicher in den Hintergrundspeicher
- $trim(blockID)$: Löscht einen gegebenen Block
- $mark(label)$: Markiert ein externes Ereignis

Die Ausführungsreihenfolge dieser Operationen kann nun wieder mithilfe von Regeln relaxiert werden. Zwei Disk Operationen o_i und o_j dürfen vertauscht werden solange eine der folgenden Bedingungen erfüllt ist:

- entweder o_i oder o_j ist ein $flush$
- o_i und o_j betreffen den selben Block
- o_i ist ein $mark$ Befehl

Der Enumerator nimmt nun den Litmus Test *P* und sendet ihn an ein Gast-System. Dieses führt *P* gegen das darunterliegende Dateisystem aus. Im folgenden werden Ereignisse $e_0, \ldots, e_n$ an die Virtuelle Disk gesendet. Als Hypervisor nimmt FERRITE diese auf und speichert sie in einer Spur t_P (Abbildung 6). Der Enumerator produziert nun alle möglichen erlaubten Vertauschungen und Präfixe, gegeben einem Disk Modell. Jeder Präfix produziert somit ein Disk-Abbild, welches zu einem Disk-Zustand nach einem Absturz gehört. Diese Abbilder können wiederum eingehängt und gegen die Prädikate des Litmus Tests *P* geprüft werden.

6.2 Bounded Model Checker

Statt Litmus Tests gegen echte Dateisysteme auszuführen, erlaubt es der *bounded model checker* einen Test *P* gegen axiomatische Spezifikationen zu testen. Die Eingabe hierfür ist ein Litmus Test *P* und ein dateisystem-spezifisches, axiomatisches Absturz-Konsistenz Modell *M*. Das Hilfsprogramm produziert nun alle Absturz-Spuren aller validen Spuren t_P gegeben *M*: $T = \{c_P | c_P$ ist eine Absturz-Spur von t_P und $M(t_P, \tau_P)\}$. Diese können im Anschluss gegen die in *P* definierten Prädikate getestet werden um zu erkennen ob das Dateisystem den zu testenden Zustand erlaubt oder nicht.

7. Aussicht

Bornholt et al. [2] haben ein Konzept (*Synthese*) vorgeschlagen, welches es erlaubt gewünschte Absturz-Konsistenz Eigenschaften zu synthetisieren. Die Idee ist, dass Anwendungsentwickler eine Anwendung *P* schreiben und die sequentielle Absturz-Konsistenz annehmen. Der Synthetisierer transformiert nun *P* durch Einfügen einer minimalen Menge von Barrieren (z.B. `fsync`). Das resultierende Programm *P'* wird sich anschließend wie *P* unter einem schwächeren Absturz-Konsistenz Modell (z.B. `ext4`) verhalten.

8. Fazit

Bei dem Problem handelte es sich um von `POSIX` unterspezifizierte Dateisystemzustände nach Abstürzen. Entwickler können somit nur raten wie man Datenintegrität erhält und gleichzeitig die Performanz der Anwendung verbessert. Die hierfür vorgeschlagene Lösung besteht aus sog. Absturz-Konsistenz Modellen um klare Richtlinien zwischen Dateisystem und Applikationen zu formalisieren. Diese bestehen aus Litmus Tests - kleinen Programmen die ein Dateisystem auf überraschendes Verhalten testen sollen - und formalen Spezifikationen, welche sich in axiomatisch mithilfe von Regeln, sowie operational über nicht-deterministische Automaten, unterteilen.

Literatur

[1] Sarita V. Adve and Hans-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101, August 2010.

[2] 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 Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 83–98, 2016.

[3] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 433–448, Broomfield, CO, October 2014. USENIX Association.

[4] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[5] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 449–464, Broomfield, CO, October 2014. USENIX Association.