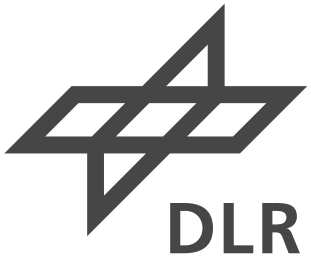
Universität Bremen



Deutsches Zentrum
für Luft- und Raumfahrt

Bachelorarbeit

in Informatik

Umgebung für automatisierte Tests von Dateisystemen auf NAND-Flash

Pascal Pieper
Mat. Nr. 2787002

Erstgutachter:	Görschwin Fey
Zweitgutachter:	Ute Bormann
Betreuer:	Fabian Greif
Abgabedatum:	24.02.2016

Erklärung

Ich, Pascal Pieper, versichere, dass ich die vorliegende Arbeit – bis auf die Betreuung durch die Abteilung Avioniksysteme des Deutschen Zentrum für Luft- und Raumfahrt (DLR) und die Arbeitsgruppe Zuverlässige Eingebettete Systeme der Universität Bremen – selbst und ohne fremde Hilfe angefertigt habe. Die benutzten Quellen und Hilfsmittel sind vollständig angegeben, Zitate sind kenntlich gemacht.

Bremen, den 24.02.2016

. . . . . .

Pascal Pieper

Zusammenfassung

Datenspeicher in Raumfahrtanwendungen sind ein Hauptfaktor für die Echtzeitfähigkeit des kompletten Systems. Ob als Langzeitspeicher für Messergebnisse oder als Hauptspeicher für Instruktionen sind sie für nahezu jedes System unabdingbar. Da in der Raumfahrt eine lange Lebensdauer und ein vorhersehbares Verhalten den Erfolg einer Mission bestimmen kann, sind benutzte Speicher um Größenordnungen von einigen 10.000 mal teurer als normale Endkundengeräte, weil sie wohldefinierte Zeitverhalten besitzen und besonders robust gegenüber Strahlung und mechanischen sowie thermischen Belastungen sind. Um den Preis für Speicher zu verringern, sollen die deutlich günstigeren NAND-Speicher aus der Massenproduktion verwendet werden und mithilfe eines geeigneten Dateisystems die deutlich höhere Fehleranfälligkeit durch Redundanz und die fehlende Echtzeitfähigkeit durch bestimmte Strategien ausgeglichen werden. In der Arbeit werden Dateisysteme vorgestellt und verglichen, die solche Aufgaben erfüllen könnten. Es wird eine Simulationsumgebung entwickelt, die Dateisysteme mit allgemeinen oder spezifischen Benutzungsprofilen testen und quantifizierbar vergleichen kann. Dazu werden Fehlerquellen von NAND-Speichern im Weltraum klassifiziert, modelliert und implementiert. Als Fallbeispiel wird je eine Implementation der Dateisysteme FAT und YAFFS in die Simulationsumgebung integriert und unter verschiedenen Aspekten verglichen. Es wird gezeigt und begründet, dass FAT ein besseres Verhältnis von Speicherverbrauch und Nutzdaten bietet, aber durch Abwesenheit von *wear leveling* und fehlerkorrigierende Codes deutlich schlechter bei starker Dateiinteraktion und unter Fehlereinflüssen abschneidet. Es wird gezeigt, warum beide Dateisysteme auf NAND-Flash nicht echtzeitfähig sind und dass eine Kombination von einem Flash Translation Layer, einem geeigneten RAID-Verbund und einem normalen Dateisystem wie ext4 dieses Kriterium erfüllen kann.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Problemstellung

Im Weltraum herrschen viele Einflüsse auf Raumfahrtsysteme. Erschütterungen, harte Strahlung und große Temperaturschwankungen wirken auf die Komponenten ein, was die Lebensdauer signifikant reduzieren kann. Der Fokus dieser Arbeit liegt auf den Datenspeichern, die einen wichtigen Teil eines Computersystems ausmachen. Es gibt in der Raumfahrt für verschiedene Aufgaben unterschiedliche Speicherarten; allerdings sind sie teurer für verhältnismäßig wenig Speicherkapazität im Vergleich zu heutigen Commercial Off-The-Shelf (COTS)-Speichern. Der Grund dafür ist, dass diese teuren Speicher eine besondere Güte besitzen, da sie robuster sind und besonderen Ansprüchen in den Fertigungsprozessen genügen. Es sind also bereits einzelne Speicher durch genaue Abläufe z.B. auf ihre Strahlungstoleranz getestet worden, und der standardisierte Fertigungsprozess erlaubt es, diese Testergebnisse für andere Speicher z.B. der selben Charge zu garantieren.

Ein sinnvolles Ziel ist es also, Datenspeicher so zu benutzen, dass sie vergleichbar günstig wie COTS-Produkte sind, aber die gleiche Sicherheit wie radiation hardened components (radhard)-Speicher garantieren. Ein Beispiel für eine günstige Speicherart ist der NAND Flashspeicher (siehe Abschnitt 2.1). Dieser ist nicht so zuverlässig und besitzt keine hohe Lebensdauer im Vergleich zu anderen Speicherarten wie Festplatten (Hard Disk Drives (HDDs)), Datenkassetten oder NOR-Speichern, allerdings ist er besonders kostengünstig. Damit können mehrere gleiche Speichereinheiten parallel benutzt werden, und immer noch weit unter dem Einkaufspreis für radhard-Speicher liegen. Natürlich muss die, für NAND-Flash typische, Fehleranfälligkeit kompensiert werden. Dies wird oft durch speziell angepasste Dateisysteme (siehe Abschnitt 2.2) erreicht, die z.B. Daten redundant auf verschiedenen Speichereinheiten verteilen und Fehler korrigieren. Bei der Auswahl eines geeigneten Dateisystems ist es wichtig, auf die Kriterien der Mission zu achten. Es könnte sich z.B. um einen Langzeitspeicher handeln, der nur selten beschrieben, aber öfter gelesen wird, oder um einen Loggingspeicher, bei dem eine große Datei sequenziell vergrößert wird. Durch die großen Unterschiede der szenarienspezifischen an ein Dateisystem ist es bisher schwer, ein passendes Dateisystem für eine gegebene Aufgabe zu finden.

1.2 Nutzen

Der Nutzen einer Umgebung, die einheitliche Tests auf vielen verschiedenen Dateisystemen ausführen kann, liegt in der einfachen Vergleichbarkeit für festgelegte Nutzungsprofile. Wenn die Anforderung darin besteht, das beste Dateisystem für z.B. einen Loggingspeicher zu finden, kann auf der Auswahl an Dateisystemen einen Test durchgeführt werden, der eine einzelne Datei sequenziell öffnet, beschreibt, und wieder öffnet. Anschließend können die entstandenen Statistiken (siehe Abschnitt 5.2) ausgewertet und verglichen werden. Auf dieser Grundlage können Dateisysteme quantitativ bewertet, und dadurch vergleichbar gemacht werden. Neben den Nutzungsprofilen sollten auch dem zu erwartenden Szenario entsprechende Fehlerraten des simulierten NAND-Speichers festgelegt werden.

Die Entwicklung eines solchen Werkzeugs ist auch nützlich bei der Entwicklung neuer Dateisysteme. Die Auswirkungen minimaler Veränderungen können gleich in dem gewünschten Szenario getestet und in einen Vergleich mit existierenden Dateisystemen gestellt werden. Mit besser auf Fehlertoleranz ausgerichteten Dateisystemen kann der Haupt- und Sicherungsspeicher in Weltraumanwendungen deutlich günstiger werden. Im Zuge dieser Bachelorarbeit soll eine solche Umgebung, Suite for Automated Testing of Filesystems On Nandflash (SATFON), entwickelt werden.

1.3 Verwandte Arbeiten

Nandsim *Nandsim* simuliert Flash bis an die Bus-Schnittstelle und verhält sich als Linux-Treiber wie ein echtes Gerät[1]. Das Programm wird häufig dazu benutzt, auf Abbilder von realen Systemen zuzugreifen und sie zu verändern, ohne die direkte Verbindung zu dem NAND-Chip zu haben. Dadurch kann das Dateisystem eingehängt und auf die Daten zugegriffen werden. Nach der Bearbeitung kann das neu entstandene Abbild auf den Chip geschrieben werden, ohne dass er seine Integrität verliert. Es wird auch für das Testen von in der Entwicklung stehenden NAND-Dateisystemen verwendet, da es auch Fehler simulieren und die Abnutzung protokollieren kann. Die simulierbaren Fehler sind statische Werte, die bestimmen, wann bestimmte Blöcke oder Pages ausfallen, und wie viele Bitkipper pro Page im gesamten Chip maximal auftreten können[2]. Fehlerursachen sind nicht vorgesehen.

NANDFlashSim Der open source Simulator „NANDFlashSim" kann Zugriffszeiten und Energieverbrauch einer oder mehrerer NAND-Speicher detailliert simulieren (Jung u. a., 2012). Der hauptsächliche Unterschied zu anderen Simulatoren ist, dass er dabei keine durchschnittliche Latenz annimmt, sondern genaue Variationen einschließt. Dies macht er unter einem Flash Translation Layer (FTL)[3], sodass genaue Konfigurationen (auch bis in die Mikroarchitektur hinein) des Zieltyps

[1] http://www.linux-mtd.infradead.org/faq/nand.html
[2] http://www.unix.com/man-page/freebsd/4/nandsim/0
[3] Siehe Abschnitt 2.1.3 auf Seite 13

ermöglicht werden, um der Wirklichkeit besonders nahe zu kommen[4]. Das Anwendungsgebiet liegt darin, die Geschwindigkeit eines Dateisystems oder von Flash-Controllern inkl. FTL und *garbage collection* zu messen und evtl. zu optimieren. Für das Analysieren von Verhalten im Fehlerfall ist dieses Werkzeug allerdings nicht geeignet, da keine tatsächlichen Daten gespeichert werden und daher Fehler oder Ausfälle nicht simuliert werden können.

DiskSim *DiskSim* ist ebenfalls ein quelloffenes Projekt. Der Nutzen dieses Simulators ist laut Handbuch (Bucy u. a., 2008) das Verständnis über die Performanz und das Erforschen neuer Architekturen von Speichermedien. Simuliert werden die Datenraten und Zugriffszeiten durch verschiedene, zuschaltbare Module wie Gerätetreiber, Busse, Controller, Adapter und die Festplatten selbst, allerdings werden zu keinem Zeitpunkt reale Daten gelesen oder geschrieben. Da der Simulator eine Schnittstelle für externe I/O-Zugriffe bietet, kann der Einfluss von Speichermedien auf die Betriebssystemperformanz mit z.B. *SimOS* (Rosenblum u. a., 1995) ausführlich getestet werden. Durch die Kapselung der Module ist es zwei Teams möglich gewesen, darauf eine detaillierte Simulation für Solid State Drives (SSDs) aufzubauen. Microsoft Research hat eine idealisierte SSD als Modul für die Simulation entwickelt[5]. Diese wird durch unterliegende Flash-Eigenschaften parametrisiert, die sich allerdings auf Schätzwerte beziehen. Außerdem werden Vorgänge wie Caching nicht unterstützt. Eine sehr viel genauere Simulation wird von Youngjae u. a. (2009) mit *FlashSim* ermöglicht. Diese simulieren die einzelnen Module einer SSD von dem Package über die Page bis zum Bussystem. Dabei werden auch FTL, Caching *garbage collection* und *wear leveling* berücksichtigt. Auch dieser Simulator ist für die Erprobung unter Fehlerbedingungen nicht geeignet, allerdings können konkrete Geschwindigkeitsannahmen gefasst werden, wenn ein Dateisystem zur Auswahl gekommen ist. Dabei könnten die I/O-Zugriffe bei den Tests mitgeschnitten werden, um sie dann in *DiskSim* wieder abzuspielen.

[4] http://camelab.org/nfs/pmwiki.php

[5] http://research.microsoft.com/en-us/downloads/b41019e2-1d2b-44d8-b512-ba35ab814cd4/default.aspx

Kapitel 2

Grundlagen

2.1 NAND-Flash

Electrically Erasable and Programmable Read-Only Memory (EEPROM)-Flash ist in den Achtzigern aus dem electrically programmable read-only memory (EPROM) entstanden (Bez u. a., 1988). Flash ist zwar pro GB teurer als herkömmlicher HDD-Speicher, ist aber durch die Bauweise deutlich resistenter gegen Erschütterungen und Magnetfelder, und ist bei vielen parallelen Lese- oder Schreiboperationen schneller. Unterschieden wird zwischen NAND und NOR; besonders NAND-Flash hat aber eine geringere Lebensdauer als herkömmliche Speicherarten (Mutlu, 2014).

2.1.1 Funktionsweise

Ein NAND-Flashspeicher speichert Daten persistent in MOSFETs, die in Reihen angeordnet sind. Diese beinhalten die Speicherinformationen für jeweils ein Bit pro Page. Eine Page ist die kleinste beschreib- und lesbare Einheit. Eine Anzahl von Pages bildet einen Block, der die kleinste löschbare Einheit bildet. Diese Eigenheiten müssen bekannt sein, um den Speicher korrekt benutzen zu können. Ein optionaler NAND-Speichercontroller kann eine Abstraktion herstellen, die Flash blockorientiert (siehe block device) darstellt. Ein Beispielaufbau wird in Abb. 2.1 dargestellt.

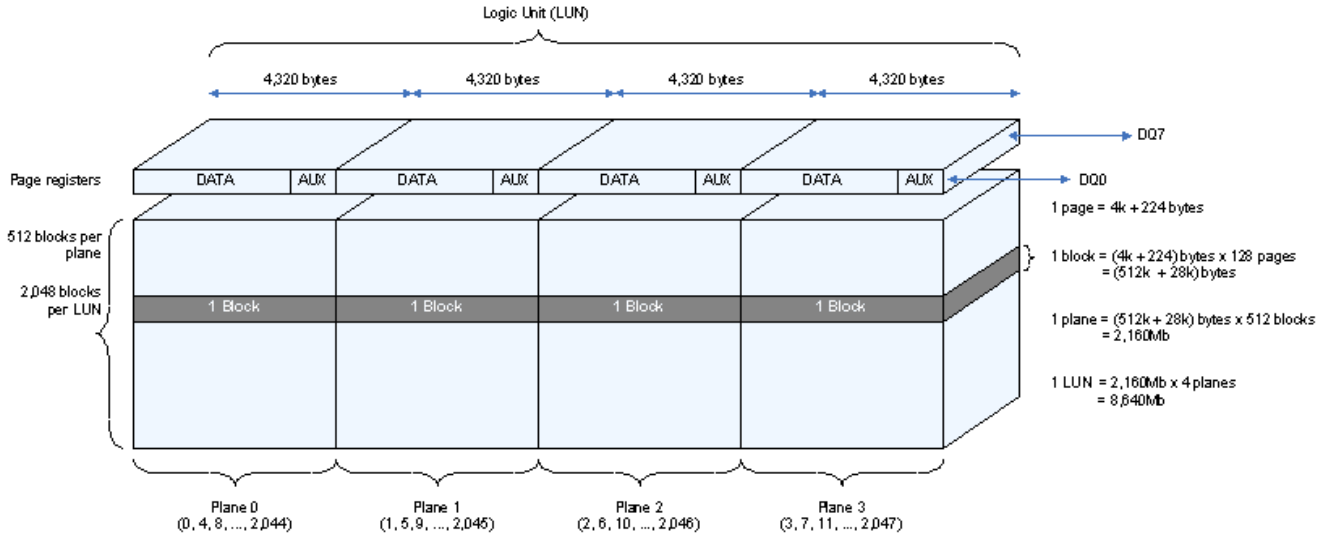
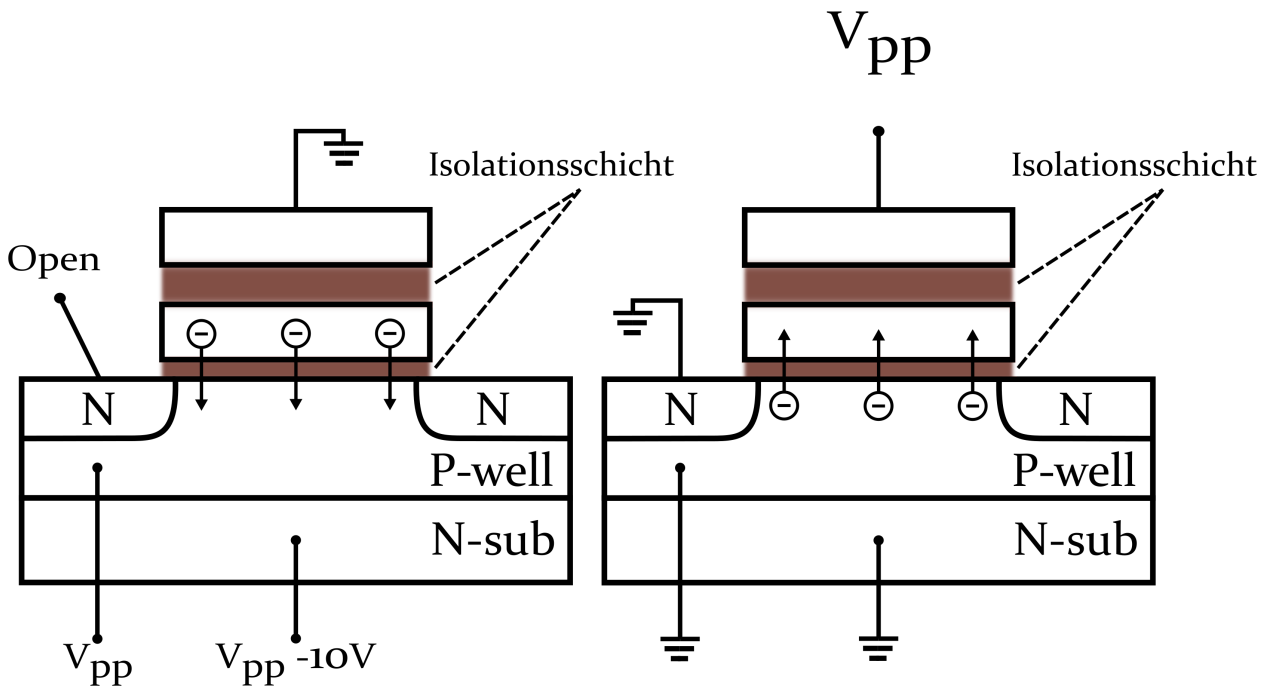


Abbildung 2.1: Einteilung eines NAND-Speicherbausteins mit beispielhaften Werten. Quelle: Jedrak (2011)

MOSFET Metal Oxide Semiconductor Field-Effect Transistors (MOSFETs) gehören im Gegensatz zu den Bipolartransistoren zu den Feldeffekttransistoren. In einem MOSFET wirkt das elektrische Feld des *gates* auf die Leitfähigkeit eines Halbleiters. Im Speicher wird der Transistor um ein isoliertes *floating gate* erweitert (siehe Abb. 2.2). Durch Tunneleffekte ist es möglich, bei Spannungen oberhalb von $12V$ (NOR) oder $18V$ (NAND) Elektronen über die Isolation hinweg in das *floating gate* einzufügen. Dort erzeugen sie das elektrische Feld, mit dem der Transistor als programmiert gilt. Fehlt diese Ladung, ist er gelöscht. Durch die NAND-Strukturierung bedeutet ,,gelöscht" logisch 1 und umgekehrt. In normalen Flashzellen wird nur ein Bit pro Gatter gespeichert (Single Level Cell (SLC)), es gibt aber auch Multi Level Cells (MLCs), die vier oder mehr Zustände (Leitfähigkeiten) speichern können, also zwei oder mehr Bit mit einem MOSFET. Dabei leidet allerdings die Lebensdauer, die Lese/Schreibzyklen reduzieren sich auf bis zu ein Zehntel eines SLC-Flash gleicher Größe (Nguyen u. Scheick, 2003).

Memory Stack Ein *memory stack* besteht bei NOR-Speicher aus einem MOSFET, und ist mit Masse und der Bit-line verbunden. Bei NAND befinden sich so viele MOSFETs in Reihe, wie es Pages gibt, wobei jeder Transistor darin zu einer anderen Page gehört (siehe Abb. 2.3). Um eine einzelne Zelle auszulesen, müssen alle anderen durch das *control gate* auf Durchgang geschaltet werden.

Page Die Page ist die kleinste beschreibbare Einheit bei NAND und NOR, für den NAND ist sie auch die kleinste lesbare Einheit. Eine Page besteht typischerweise aus 1024 Byte Nutzdaten und einer 32 Byte großen Out Of Band Area (OOBA) oder den Vielfachen von ihnen. Dabei ist für den NAND-Speicher unerheblich, ob die OOBA-Regionen auch für Nutzdaten benutzt werden oder Metainformationen



(a) Programmieren (b) Löschen

Abbildung 2.2: Aufbau eines einzelnen MOSFETs. Gezeigt wird die Art, wie ein Bit durch Fowler-Nordheim-Tunneln gesetzt oder gelöscht wird. Neu erstellt, Quelle: Nguyen u. a. (1998)

enthält. In der Regel werden dort Informationen wie zum Beispiel Error Correcting Code (ECC)- oder Error Detecting Code (EDC)-Daten und *bad block marker* von einem Dateisystem oder einem Controller gespeichert. Im Gegensatz zum NOR-Flash kann eine Page je nach NAND-Typ nur wenige Male bis ein einziges Mal ohne einen Löschvorgang beschrieben werden (Woodhouse, 2008), bevor Inkonsistenzen auftreten.

Block Ein Block besteht aus typischerweise 128 Pages (also 128KiB Nutzdaten + 4KiB OOBA), und ist die kleinste löschbare Einheit.

Plane Die Plane beinhaltet in der Regel 256 Blöcke und einen (volatilen) Zwischenspeicher pro Block in der Größe einer Page. Bei einem Lese- oder Schreibvorgang werden die Daten dort aus der entsprechenden Page zwischengespeichert. Um zusammenhängende Lese-/Schreiboperationen zu beschleunigen, werden die physikalischen Nummern der Blöcke gleichmäßig über alle Planes verteilt, also ist z.B. bei n Planes jeder $n$te Block in der ersten Plane.

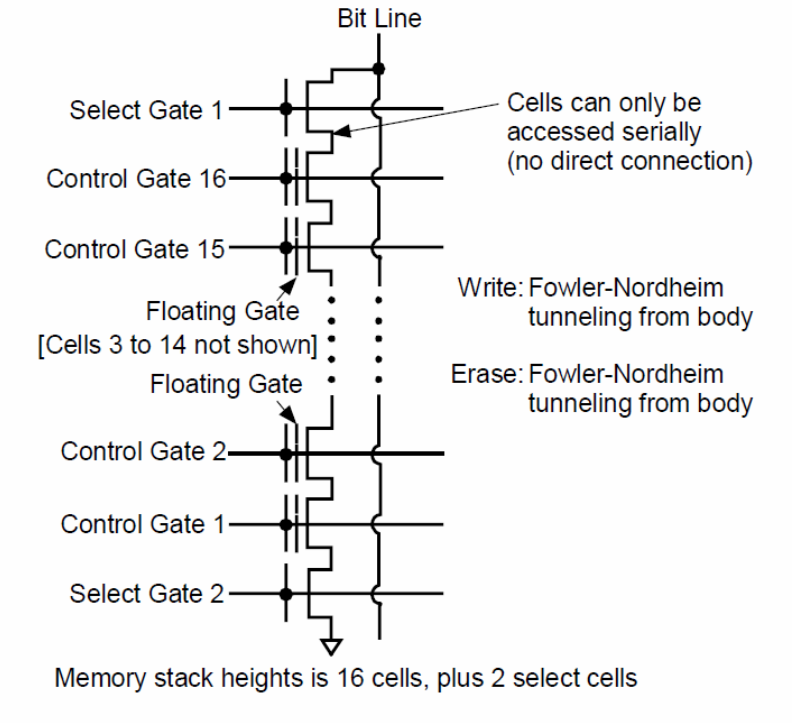


Abbildung 2.3: NAND Struktur; Serielle Anordnung der MOSFETs, mit *select*-Gattern. Quelle: Nguyen u. Scheick (2003)

Logic Units Eine *logic unit* oder manchmal auch *„die"* besteht i.d.R. aus 4 Planes, bei denen die Blocknummern abwechselnd in den Planes verteilt sind.

Package Das *package* ist der endgültige NAND-Chip, der beliebig viele *logic units* enthalten kann.

Peripherie Ein Flash-Chip wird in der Regel bei $3,3V - 5V$ betrieben. Da aber für die Programmierung und das Löschen höhere Spannungen benötigt werden, gibt es eine Ladungspumpe, die verschiedene Spannungen bereitstellt.

Interfacecontroller Der Interfacecontroller befindet sich mit auf dem Chip und stellt die I/O-Schnittstelle zur Verfügung. Er interpretiert z.B. die Spannungspegel der MOSFETs oder steuert die Dauer und Intensität von Löschvorgängen. Er beinhaltet dazu eine state machine, die das Protokoll der Busanbindung implementiert und die internen Prozesse steuert, sowie eine meist als Ladungspumpe gebaute Spannungsversorgung, die die genauen Lösch- und Programmierspannungen bereitstellt.

Speichercontroller Neben den normalen Speicherbausteinen kommt in der Praxis wie z.B. bei USB-Speichern ein NAND-Controller dazu. Dieser wird auch als *glue logic* bezeichnet. Seine Aufgabe besteht darin, die unterliegende Flash-Struktur zu verbergen, damit der Speicher als block device erscheint. Dadurch können Datei-

systeme wie z.B. FAT (Abschnitt 3.4) eingesetzt werden, die nicht für die Verwendung von NAND-Flash entwickelt wurden.

Der Speichercontroller wird in SATFON absichtlich weggelassen (Siehe Abschnitt 4.2), da diese Aktionen auch von Dateisystemen unterstützt werden (siehe Abschnitt 2.1.3), und deren Abschneiden das Hauptinteressengebiet der Simulation ist.

2.1.2 Fehlerursachen

Um eine Simulation zu entwickeln, die möglichst genau die Realität abbilden soll, müssen zuerst die Fehlerursachen und deren Auswirkungen klassifiziert werden. Diese Fehlerklassen dienen als Grundlage für die Implementation.

Abnutzung Ein NAND-Flash hat eine Lebensdauer von i.d.R. $10.000 - 100.000$ Lösch- und Schreibzyklen im Gegensatz zu $100.000 - 1.000.000$ Zyklen bei NOR-Flash, bedingt durch eine höhere Programmier- und Löschspannung. Die Abnutzung entsteht durch die Art der Programmierung des *floating gate* MOSFETs. Bei jeder Programmierung werden durch die untere, oft nur 200 Å dicke Isolationsschicht (siehe Abb. 2.2) Elektronen getunnelt. Dieser Vorgang zerstört langsam die Oxidschicht, bis sie nicht mehr ausreichend isoliert und Ladung entweichen kann.

Strahlung Zusätzlich zu dem o.g. Alterungsprozess gibt es eine Anfälligkeit gegenüber Strahlung in den Speicherregionen (Nguyen u. Scheick, 2003), in der state machine des Controllers und in der Spannungsversorgung (vgl. genaue Auflistung in Tabelle 2.1). Die vergleichsweise anfällige interne Spannungsversorgung ist laut Nguyen u. a. (1998) der Hauptfaktor einer geringen Strahlungsresistenz (Total Ionizing Dose (TID)) gemessen in kRad. Um die Lebensdauer zu erhöhen ist empfohlen, eine externe Versorgung zu wählen, da große Strukturen weniger anfällig sind. Weiterhin wird gezeigt, dass das Fortschreiten der TID längere Lösch- und Leseverzögerungen sowie einen höheren Stromverbrauch mit sich führt, da die benötigten Programmier- und Löschspannungen aufgrund von Leckströmen schlechter erreicht werden. Im Gegensatz zu Strahlungsfehlern durch eine TID gibt es auch Probleme, die von Single Event Effects (SEEs), also einzelnen, energiereichen Partikeln, hervorgerufen werden können. Dabei wird zwischen einmaligen (Single Event Upsets (SEUs)) und potentiell dauerhaften (Single Event Latchups (SELs)) Fehlern (vgl. O'Bryan u. a. (2011)) unterschieden. Der SEU kann zum Beispiel ein Bitkipper im Speicher sein, aber auch in der state machine des Interfacecontrollers. Ein solcher Fehler in der *statemachine* kann dazu führen, dass Zugriffe länger dauern oder das Gerät nicht mehr reagiert. Letzteres kann aber durch einen Neustart behoben werden. SELs sind schwieriger zu handhaben, da sie dauerhafte Fehler hervorrufen. Dies kann einen MOSFET oder eine ganze Region des vollständigen Chips dazu bringen, dauerhaft falsche Werte zu produzieren, oder es wird ein parasitärer Thyristor gezündet. Dabei leiten unerwünscht P- und N-dotierte Zonen untereinander und erzeugen so einen Kurzschluss. Dies führt ohne Sicherungsmechanismen zur thermalen

Zerstörung des Chips. Als Abhilfe zu parasitären Thyristoren kann die Fertigungstechnik geändert werden (z.B. BiCMOS) oder im Nachhinein externe Überstromsensoren auf dem Endprodukt eingebaut werden, die sofort nach der Erkennung eines Kurzschlusses die Stromversorgung abschalten. Falls rechtzeitig ausgeschaltet wurde, kann der Chip evtl. mit geringen Schäden wieder in Betrieb genommen werden, ansonsten ist das Gerät nicht mehr verwendbar.

Fehlerklasse	*Mögliche Auswirkung*	*Mögliche Ursache*
Fertigungs-toleranzen	Ab Werk ausgefallene Pages oder Blöcke	Leitungsunterbrechungen
	Permanente Bitfehler	Isolationsschicht besitzt nicht korrekte Dicke Ungewollter Kontakt zwischen Leitungen
Alterung & Abnutzung	Häufung von Bitfehlern in einzelnen Blöcken	Dissipation von Elektronen Abnutzung der Isolationsschichten
	Geringere Schreib- und Löschgeschwindigkeit	Interne Spannungsversorgung erreicht optimale Programmier- und Löschspannung durch interne Lecks nicht mehr Kontaktverlust durch Temperaturschwankungen oder mechanischen Belastungen
Erhöhte TID	Bitflips im Speicher	Erhöhte Empfindlichkeit der Speicherzone durch Partikeleinschlag
	Erhöhter Stromverbrauch	Durch Ionisierung entstandene parasitäre Verlustleistungen
	Lese- Schreib- und Löschfehler oder starke Verzögerung der Ausführung	Erhöhte Empfindlichkeit der Register der internen state machine durch Partikeleinschlag
	Aussetzen des Speichercontrollers	Veränderung der Register der internen state machine durch erhöhte Empfindlichkeit durch Partikeleinschlag
	Komplettes Versagen	Zerstörung der internen Logik durch Partikeleinschlag oder Ionisierung
SEU	Bitflips im Speicher	Entladung oder Aufladung der floating gates
	Lesefehler	Veränderung des Lesepuffers oder der state machine durch Partikeleinschlag
	Aussetzen des Betriebes	Deadlock der state machine durch veränderte Register im Interfacecontroller
	Verzögerung von Operationen	Veränderung der Register der state machine
SEL	Permanente Bitfehler	Zerstörung der Isolationsschichten durch Partikeleinschlag
	Misserfolg von Schreib- oder Löschoperationen auf einzelnen Blöcken	Zerstörung von Leitungsverbindungen oder Schaffung von Leckströmen
	Komplette Zerstörung der Komponente	Zündung eines parasitären Thyristors oder Zerstörung von Verbindungen durch Partikeleinschlag

Tabelle 2.1: Überblick von Fehlerklassen, ihren Auswirkungen auf den laufenden Betrieb und ihren mögliche Ursachen

2.1.3 Gegenmaßnahmen

Besonders in gebräuchlichen SD/MMC oder USB-Speichern werden Sicherungsalgorithmen meistens in einem externen Flash-Controller untergebracht. Dieser übernimmt zum Beispiel das Verwalten der OOBA-Daten, also das *bad block handling* und die Fehlerkorrektur/-erkennung. Diese Aufgabe erledigt der sogenannte Flash Translation Layer (FTL). Wird durch ihn bei einer Leseoperation erkannt, dass die Daten häufiger nicht mit dem ECC übereinstimmen, wird der betroffene Block als ,,*Bad*" markiert, und nicht mehr für Operationen benutzt. Dies passiert allerdings transparent für den Anwender; will er also z.B. auf den logischen Block *2* zugreifen, und der Block defekt ist, leitet der FTL diese Anfrage auf versteckte oder freie Blöcke um (Park u. a., 2009). Oft enthält er auch einen *wear leveler*, der oft beschriebene (logische) Blöcke immer wieder auf unterschiedliche, wenig abgenutzte physikalische Blöcke abbildet. Unterschieden wird zwischen statischem und dynamischem *wear leveling*. Dynamisches *wear leveling* bildet diese nur auf unbenutzte Blöcke ab, die statische Methode hingegen bezeichnet, dass ein neu angeforderter Block auch aus dem Pool der benutzen Blöcke kommen darf. Diese Variante ist zwar komplexer und erfordert bei einem besetzten Block pro Umleitung mehrere Schreibzugriffe, allerdings können die Abnutzungserscheinungen deutlich besser verteilt und damit die Lebensdauer, je nach Benutzung, um den Faktor 4 im Vergleich zur dynamischen Methode erhöht werden (Micron Technology, 2008). Das Löschen von Daten in einer physikalischen Page wird dabei nicht sofort ausgeführt, um Löschvorgänge zu sparen. Erst, wenn der gesamte Block sog. *dirty* (veraltete) Pages enthält, oder der Speicherplatz knapp ist, wird er gelöscht. Dafür ist ein *garbage collector* notwendig, der bei Mangel an freien Blöcken die (im OOBA) als *dirty* markierten Pages zusammenfasst und löscht. Bei einem solchen Controller ist es möglich, Dateisysteme sinnvoll einzusetzen, die keine Kenntnis von der unterliegenden Flashstruktur besitzen (vgl. Lin u. a. (2006)). Dabei ist zu beachten, dass die meisten FTLs durch den *garbage collector* nicht echtzeitfähig sind. Besonders wenn die freien Blöcke knapp werden, kann es zu großen Verzögerungen bei Schreibvorgängen kommen (Qin u. a., 2012).

Eine Methode, Daten im Allgemeinen länger zu konservieren, ist, sie auf mehrere Speichereinheiten zu kopieren. Dabei können oben genannte Strategien damit kombiniert werden, um die Lebensdauer weiter zu erhöhen. Diese Idee wendet auch Redundant Array of Independent Disks (RAID) an, welches in den 80er Jahren zuerst ,,Redundant Array of Inexpensive Disks" genannt wurde. Es war die Antwort auf die damals steigenden Computergeschwindigkeiten, aber stagnierende HDD-Kapazitäten und Geschwindigkeiten. Durch RAID können z.B. mehrere Festplatten zu einer virtuellen Platte bei deutlicher Geschwindigkeitsverbesserung auf Kosten der Lebensdauer zusammengeschlossen werden. Allerdings ist RAID keine eigenständige Implementation, sondern nur eine Technik. Daher gibt es oft Unterschiede zwischen verschiedenen Herstellern von Hard- oder Software-RAID-Implementationen. Hardware RAID bedeutet, dass es für jede Speichereinheit einen extra Chip gibt, der zusammen mit den anderen den RAID-Controller bildet. Die-

ser bildet die unterliegenden Speichereinheiten je nach Einstellung transparent für das Betriebssystem ab. Software RAID wird oft als Treiber geliefert, hierbei muss die Central Processing Unit (CPU) sämtliche Berechnungen durchführen, was einen erheblichen Einfluss auf die Performanz haben kann. In 1988 wurden die ersten 5 Modi veröffentlicht, und von dort an wurden viele Kombinationen entwickelt.

RAID-0 Die Kernidee bei RAID-0 ist, Daten eines virtuellen Blockes gleichmäßig auf viele gleichgroße Datenträger zu verteilen. Die virtuelle Blockgröße hängt von der Anzahl der Datenträger ab, und ist die Summe der einzelnen physischen Blockgrößen. Dadurch steigt die Kapazität des virtuellen Datenträgers an. Ein Nebeneffekt ist, dass Lesen und Schreiben sehr viel schneller wird, da jede einzelne Einheit nur ihren Anteil des Blockes kopieren/lesen muss. Dadurch sinkt die Mean Time To Failure (MTTF) deutlich, da bei dem Ausfall einer einzelnen Speichereinheit die gesamten Daten verloren gehen.

$$MTTF_{Group} = \frac{MTTF\ of\ a\ Single\ Disk}{Number\ of\ Disks\ in\ the\ Array}$$

(Patterson u. a., 1988)

RAID-1 Bei RAID-1 werden alle Blöcke gespiegelt. Es wird davon ausgegangen, dass fehlerhafte Blöcke erkannt werden, z.B. durch Fehlererkennung im Dateisystem, Datenträger, oder durch kompletten Ausfall einer Speichereinheit. Die MTTF steigt bei jedem zusätzlichen Speicher:

$$MTTF_{RAID} = \frac{(MTTF_{Disk})^2}{n_{Disks} * n_{Disks\ in\ a\ group} * MTTR}$$

(Patterson u. a. (1988), vereinfacht)
wobei MTTR als ,,Mean Time To Repair" und n_{Disks} als ,,Number of Disks in Array" definiert wird. Zusammengefasst können also alle Datenträger bis auf den Letzten ausfallen, ohne dass die Daten verloren werden. Dafür beträgt die Gesamtkapazität nur die Größe der kleinsten Speichereinheit.

RAID-5 In diesem Modus werden mindestens drei Speichereinheiten benötigt, wobei der logische Speicherplatz bei n Festplatten $n-1$ mal die Größe der kleinsten Speichereinheit beträgt. Über alle Einheiten werden Paritätsdaten verteilt, die sich aus jeweils einem logischen Block errechnen. So kann genau ein Ausfall kompensiert werden, unabhängig davon, welche Festplatte die Fehler aufweist. Diese Anordnung beschleunigt das Lesen ähnlich wie RAID-0, aber ist im Schreiben langsamer als RAID-0, da die Grundlagen für die Paritätsdaten gelesen, die Parität berechnet und die neuen Daten geschrieben werden müssen.

RAID-10 Dieser RAID-Modus ist die Kombination von -1 und -0. Das bedeutet, dass z.B. bei dem Minimum von 4 Datenträgern jeweils ein Paar zu einem RAID-1 verbunden, und diese dann als RAID-0 erweitert werden. Die Speicherkapazität beträgt die Hälfte der verfügbaren Speichereinheiten, und es kann pro untergeordnetem RAID-1 ,,Fuß" eine Einheit ausfallen, bevor Datenverlust

eintritt. Bei 4 Festplatten tritt der Datenverlust also bei dem Ausfall von zwei (ein ,,Fuß") oder drei Datenträgern ein.

Die RAID-Modi sind ursprünglich für HDDs gedacht, die eine deutlich geringere Fehlerwahrscheinlichkeit haben. Ist bei einer Festplatte ein Sektor kaputt, wird dieser in der Regel durch den Speichercontroller transparent durch einen Neuen ersetzt. Um RAID auf NAND-Flash zu benutzen, müsste entweder ein abstrahierender Controller (FTL) eingesetzt werden, oder das System muss gleiche Blöcke an unterschiedlichen Positionen auf verschiedenen Speichereinheiten verwalten können. Bei RAID-Modi, die nativ mit Parityblöcken arbeiten, ist zusätzlich eine Verringerung der Lebensdauer der Flashspeicher möglich (Lee u. a. (2011), Kim u. a. (2013)), da jede Schreibanfrage einen Lesezugriff und zwei Schreibzugriffe auslöst. Bei jedem Sicherungssystem, ob nun durch Redundanz oder durch bessere ECCs, ist regelmäßiges *disk scrubbing* wichtig (Gao u. a., 2010). *Scrubbing* bezeichnet die Überprüfung aller Daten auf Konsistenz, und sollte häufig durchgeführt werden. Im Test mit einer RAID-6 Anordnung ist der Unterschied der Lebensdauer zwischen täglichem und jährlichem *scrubbing* signifikant (bis zu 1000 Mal größere Lebensdauer bei HDDs). Ohne die geringen Abstände würden Fehler erst erkannt werden, wenn ein Zugriff auf diesen Teil geschieht. Wenn sich Fehler akkumuliert haben, können sie nicht wiederhergestellt werden.

2.2 Dateisysteme

Ein Dateisystem regelt, wie Dateien zu finden sind, und verwaltet die Art, wie sie auf den zugrundeliegenden Speichern verändert werden können. Dabei abstrahieren Dateisysteme die Position der Dateien von logischen Speicheradressen auf Pfade wie Ordner. Dabei können bei manchen Dateisystemen Objekte auch verteilt auf anderen, über ein Netzwerk erreichbaren, Speichern liegen. Oft werden zu einer Datei verschiedene Metadaten gespeichert, wie zum Beispiel Erstellungsdatum, Besitzer und Rechte. Ein *journaling* Dateisystem protokolliert Aktionen wie Löschen, Verschieben oder Erstellen an einer besonderen Position. Fällt bei einem Schreibvorgang die Stromversorgung aus, wird zwar das Dateisystem in einem inkonsistenten Zustand hinterlassen, kann aber wieder durch die Logdatei repariert werden.

2.3 POSIX

Das Portable Operating System Interface X (POSIX) ist ein Standard (Cooperation of open taskgroups, 2009), der viele Interaktionen mit einem Betriebssystem spezifiziert. Da es in dieser Bachelorarbeit um Speicher und Dateisysteme geht, wird nur der Teil vorgestellt, der sich mit Dateien befasst. Mit diesen Funktionen sind alle Interaktionen mit Dateien möglich. Die markierten Basic I/O Funktionen gehören zu dem minimalen Funktionsumfang, um mit Dateien umgehen zu können.

Rückgabewert	*Funktion und Funktion*	*Basic I/O*
int	access(const char *pathname, int mode); Zugriffsrechte einer Datei abfragen	
int	open(const char *pathname, int flags); Datei öffnen und eventuell erstellen	✓
int	creat(const char *pathname, mode_t mode); Datei öffnen und eventuell erstellen	
int	close(int fd); Dateideskriptor schließen	✓
ssize_t	read(int fd, void *buf, size_t count); Aus Dateideskriptor lesen	✓
ssize_t	write(int fd, const void *buf, size_t count); In Dateideskriptor schreiben	✓
int	fcntl(int fd, int cmd); Dateideskriptor modifizieren	
int	fstat(int filedes, struct stat *buf); Eigenschaften und Zustand einer Datei abfragen	✓
off_t	lseek(int fildes, off_t offset, int whence); Die Schreib-/Leseposition verändern	✓
int	dup(int oldfd); Dateideskriptor duplizieren	
int	dup2(int oldfd, int newfd); Dateideskriptor duplizieren	
int	pipe(int filedes[2]); Pipe erstellen	
int	mkfifo (const char *pathname, mode_t mode); Eine named pipe erstellen (Spezielle Datei)	
mode_t	umask(mode_t mask); Dateierstellungsmaske setzen	
FILE *	fdopen (int fildes, const char *mode); Dateideskriptor mit einem Datenstream verbinden	
int	fileno(FILE *stream); Dateideskriptor eines streams zurückgeben	

Tabelle 2.2: Eine Liste von POSIX-Befehlen, die zur Interaktion mit Dateien benutzt werden. Die markierten Funktionen gehören zu dem Mindestmaß an I/O. Frei nach http://www.kompf.de/cplus/posixlist.html

Kapitel 3

Behandelte Dateisysteme

3.1 Anforderungen an ein NAND-Dateisystem

Die allgemeinen Maßnahmen, die ein Dateisystem bei Flash treffen sollte, sind denen in Abschnitt 2.1.3 sehr ähnlich. Neben des Mindestmaßes an I/O-Funktionen (siehe Abschnitt 2.3) ist die wichtigste Eigenschaft ein ECC, der gespeicherten Daten validiert. Um die Lebensdauer zu erhöhen, sollten Schreibzugriffe über den gesamten Speicher verteilt werden (*static wear leveling*). Dies zieht die Notwendigkeit einer *garbage collection* mit sich, die oft bei Erreichen der maximalen Speicherkapazität die Performanz negativ beeinflusst. Oft erfüllt der *garbage collector* keine Echtzeitanforderungen (Qin u. a., 2012), die in Raumfahrtanwendungen durchaus notwendig sein können. Um Inkonsistenzen bei plötzlichen Stromausfällen zu vermeiden, sollte *journaling* betrieben werden (siehe Abschnitt 2.2). Weiterhin sollte es ein System geben, mit dem dieses Dateisystem (oder eine Abstraktion darüber) mehrere Speicher zu einem zusammenfasst, um die Konsistenz und Verfügbarkeit zu erhöhen, besonders weil es bei zunehmendem Fortschritt der TID wahrscheinlicher wird, dass eine ganze Speichereinheit ausfällt (siehe Abschnitt 2.1.2). Ein gutes Dateisystem sollte danach versuchen, die Speichereinheit wiederherzustellen (Neustart der Komponente, anschließender Testdurchlauf) oder ohne die Einheit weiterhin funktionieren können. Zu möglichen Optimierungen zählt die Kompression von Dateien, das Speichern mehrerer Objekte auf einer Page und der effektiven Verteilung von Dateien im Bezug auf Zugriffs- oder Transfergeschwindigkeit. Diese Funktionen können auch auf verschiedene Module verteilt werden.
Zusammengefasst sollte ein Dateisystem mindestens heiße, eher jedoch eine Kombination von heißer und kalter Redundanz auf verschiedenen physikalischen Chips anbieten können, es sollte einen erweiterten Fehlersicherungsmechanismus innerhalb der Pages oder Blöcke benutzen, *journaling* betreiben, und ohne einen FTL auskommen können. Wenn auf dem System zeitkritische Daten gespeichert und abgerufen werden müssen, ist eine Echtzeitfähigkeit sinnvoll.

3.2 Vergleich existierender Strategien

3.2.1 Dateisysteme

JFFS Eines der verbreitetsten Flash-Dateisysteme ist *JFFS* und steht für *Journaling Flash File System* (Woodhouse, 2008). Es ist zuerst für NOR-Flash entwickelt worden, ist aber auch kompatibel zu NAND. Die Hauptintention war es, ein stabiles, sicheres und langlebiges Dateisystem zu entwickeln. In der zweiten Version *JFFS2* ist ein effektiverer *garbage collector* und Kompression implementiert worden. Bei diesem Dateisystem wird allerdings nur ein EDC benutzt, es können also keine Bitfehler korrigiert werden.

NAFS Das *NAND flash memory Array File System* ist von Park u. Kim (2011) entwickelt worden. Es unterstützt mehrere NAND-Speicher um die Speicherkapazität zu erhöhen. Der Fokus der Arbeit liegt auf der Lese- Schreibgeschwindigkeit, geringeren Dauer des *mounting* bei großer Anzahl von Dateien (bis zu 375% schneller im Vergleich zu *JFFS2*) und auf der Benutzung mehrerer Speicher im RAID-0-ähnlichen Verbund, ohne auf abstrahierende Flashcontroller angewiesen zu sein[1]. Es werden auch, neben den normalen OOBA-ECC, ECC-Daten in ganzen Blöcken über alle Speicher verteilt, um eine RAID-5 ähnliche Fehlertoleranz zu entwickeln. In dem Paper wird nicht näher auf die Anzahl verkraftbarer Ausfälle eingegangen, aber durch die Verwandtschaft zu RAID-5 wird ein Laufwerksausfall kompensiert werden können. Dieses Dateisystem betreibt kein *journaling*.

YAFFS YAFFS steht für *Yet Another Flash File System* und ist von der Aleph One Corporation (Manning, 2010) entwickelt worden. Ziele der Implementierung sind Portabilität, Einfachheit und Stabilität. Durch die großzügige Lizenz wird YAFFS mittlerweile in vielen Geräten wie z.B. einigen Android Smartphones benutzt und ist deshalb sehr weit verbreitet. Es wurde schon für Linux, WindowsCE, eCos und viele RealTime Operating Systems (RTOSs) implementiert, da durch den *yaffs direct* Modus wenig externe Abhängigkeiten vorhanden sind. Es betreibt *journaling* und besitzt eigene ECC-Algorithmen, die im OOBA auf Wunsch benutzt werden.

SMARTFFS SMARTFFS ist ein Dateisystem von Chen u. a. (2006) und wurde mit dem Ziel entwickelt, schneller als andere Dateisysteme zu *mounten* und wenig Arbeitsspeicher zu benutzen. Dabei erzielt es bei großen Dateien ab 256 KiB bessere Transferraten als *YAFFS* und *JFFS*. Journaling wird nicht unterstützt, und es kann Fehler in Pages erkennen, aber nicht korrigieren.

[1] Bei dem Spiegeln von flashbasierten Speichereinheiten ist eine Abstraktion notwendig, da bei Ausfall eines Blockes der Inhalt verschoben werden muss, er aber bei den anderen Speichereinheiten an der gleichen Stelle bleibt.

CFFS Das *Core Flash File System* basiert auf *YAFFS* (Lim u. Park, 2006). Das Hauptaugenmerk liegt auf schnelleren Lesezugriffen und einem geringeren *garbage collection overhead*. Das bedeutet, dass bei Erreichen der Speicherkapazität freier Platz schneller sortiert werden kann. Von einer Fehlererkennung oder -korrektur sowie *journaling* wird nicht gesprochen, und das *wear leveling* ist laut (Park u. Kim, 2011, S. 2) schlechter als z.B. bei *YAFFS*, da die Adressen der Indexblöcke in dem ersten Block gespeichert werden. Das bewirkt, dass dieser Block häufiger gelöscht werden muss, und somit schneller zerstört ist. Der Ausfall bedeutet einen vollständigen Verlust der Daten.

NAMU Das *NAnd flash MUltimedia filesystem* von Park u. Kim (2009) ist speziell für große Dateien entwickelt worden. Durch die Möglichkeit, in einer einzelnen Metadatenbeschreibung (*index page*) mehrere aufeinanderfolgende Pages zu einer Datei zu bestimmen, ist der Zugriff auf große Dateien schneller und benötigt weniger Metadaten. Da sequenzielles Lesen und Schreiben bei NAND-Flash 20% bis 30% schneller als wahlfreier Zugriff ist[2], wird die Übertragungsgeschwindigkeit verbessert, wenn die Pages einer Datei hintereinander stehen. Die Annahme von Multimediadateien beeinflusst auch die Strategie der *garbage collection*. Weil Multimediadateien selten geändert werden, wird nicht auf Blockebene, sondern auf Dateiebene aufgeräumt. Wenn eine Datei verhältnismäßig viele unsaubere Blöcke besitzt, wird die ganze Datei verschoben, und der Platz freigegeben. Für *wear leveling* werden im Leerlauf auch selten benutze Blöcke erkannt, und die dazugehörigen Dateien in häufig benutze Gebiete verschoben (*static wear leveling*). Das Dateisystem betreibt *journaling*, allerdings ist ein ECC nicht vorgesehen.

RTFFS Jain u. Lee (2006) entwickelten das *Real-Time Flash File System*, welches Echtzeitanforderungen entsprechen soll. Der spezielle Ansatz ist, dass Daten redundant auf jeweils zwei Speichereinheiten (*banks*) gehalten werden. In diesem *,,set"* ist immer eine Bank im *,,read"*, und die Andere im *,,write"* Modus. Wenn die Anzahl der *,,dirty"*, also veralteten und nicht mehr gebrauchten, Pages zu groß wird, wird der *garbage collection* Algorithmus aktiv, wonach die Bänke ihren Zustand wechseln. Dieses Modell benötigt allerdings einen nicht volatilen konventionellen Zwischenspeicher pro Bank. Dort werden Schreibanfragen zwischengespeichert, bis sich der Zustand auf *,,write"* ändert. Auf dem jeweils anderen Block (der schon im Schreibzustand ist) kann der Schreibzugriff zusammen mit der *garbage collection* simultan abgearbeitet werden. Obwohl das Dateisystem *journaling* betreibt, fehlt eine Fehlererkennung für Pages im OOBA, mit der Daten durch die sonst unbeachtete Redundanz wiederhergestellt werden können. In diesem Fall könnte ein echtzeitfähiger Algorithmus die Wiederherstellung auf Basis der anderen Bank oder ihrem Schreibcache, sowie eventuell weiteren Sets, erlauben.

[2] (Park u. Kim, 2009); u.A. da aufeinanderfolgende Blöcke in benachbarten Planes liegen.

UBIFS Das *Unsorted Block Images File System* ist von Nokia und der Universität Szeged entwickelt worden[3]. Im Gegensatz zu anderen Dateisystemen arbeitet es nicht direkt auf dem Flash (Memory Technology Device (MTD)), sondern auf dem Abstraktionstreiber *UBI*[4]. Es betreibt *journaling* und besitzt, durch *UBI*, fehlerkorrigierende Codes.

RCFFS Das *Reliable Compressing Flash File System* wurde von Kang u. Miller (2009) entworfen und benutzt erweiterte ECCs. Im Gegensatz zu den meisten anderen Dateisystemen trennt es die Fehlererkennung und -korrektur. Der limitierte Platz im OOBA wird nur für die Erkennung benutzt, was eine deutlich gestiegene Erkennungswahrscheinlichkeit bedeutet; bei k Bit Fehlererkennung kann sie Fehler beliebiger Größe mit der Wahrscheinlichkeit $1 - 2^k$ erkennen. Die Fehlerwiederherstellung geschieht durch Paritätsdaten in Pages und Blöcken. Wenn weniger Pages Fehler enthalten, als es Paritäts-Pages in dem Block gibt, können diese so wiederhergestellt werden. Die Anzahl von Paritäts-Pages pro Segment kann bei der Formatierung und zur Laufzeit (durch ein *remount*) festgelegt werden. Wenn es mehr fehlerhafte als redundante Pages in einem Block gibt, müssen alle Blöcke in einem Segment ausgelesen und der gesamte Block wiederhergestellt werden. Dies geschieht mit Reed-Solomon-Codes in Parity-Blöcken pro Segment, wobei auch die Anzahl von Blöcken in einem Segment frei wählbar ist. Durch diese Freiheit kann der Kompromiss zwischen Sicherheit und Speicherplatzverbrauch sehr dynamisch an die Aufgabe angepasst werden. Unimplementiert, aber dokumentiert, bietet Kang u. Miller (2009) einen *garbage collector* mit *consistency checker* an. Dieser überprüft zusätzlich zu üblichen Aufgaben eines garbage collectors im Leerlauf oder in einem festgelegten Zeitintervall sequenziell alle Pages des Speichers entweder nur auf gültige Fehlerkorrekturcodes (ohne Fehler in den Nutzdaten zu beheben), oder überprüft zusätzlich auch die Daten, um sie u.U. zu korrigieren. Dieses *sweeping* kann nach Gao u. a. (2010) die Lebensdauer signifikant erhöhen, da sich bei NAND-Flash Fehler auch ohne Zugriffe akkumulieren können. Die Lebensdauer eines einzelnen Chips würde, im Vergleich zu konventionellen Dateisystemen, vermutlich sinken, da für jeden Schreibzugriff auf eine Page sowohl die Paritätspages als auch der Paritätsblock innerhalb eines Segmentes beschrieben werden muss, und es keinen *wear leveling* Algorithmus gibt. Seit 2009 wird das Dateisystem nicht mehr weiterentwickelt.

3.2.2 RAID und ähnliche Techniken

Flash-Raid Im u. Shin (2011) (wie auch Lee u. a. (2011)) stellen eine RAID-5 Variante vor, die Paritätsdaten verzögert schreiben kann. Dies bewirkt, dass zusätzlich benötigte Schreib/Lesezugriffe erst nach einer zusammenhängenden Übertragung stattfinden, und nicht bei jedem atomaren Zugriff auf eine Page. Mit dieser

[3] http://www.linux-mtd.infradead.org/doc/ubifs.html
[4] http://www.linux-mtd.infradead.org/doc/ubi.html

Strategie können laut Im u. Shin (2011) bis zu 33% der Löschzugriffe eingespart werden. Der Nachteil ist, dass sie einen unterliegenden FTL benötigen.

Diesen Ansatz verfolgen auch Lee u. a. (2009), allerdings verzögern sie die Parität noch länger. Erst, wenn die Speichereinheit im Leerlauf ist, oder der Zwischenspeicher voll ist, werden Paritäten geschrieben. Dies hat eine bessere Abnutzungsverteilung zur Folge und vergrößert angeblich die Lebensdauer um bis zu 44% im Vergleich zu reinem RAID-5 auf einem FTL. Mit der höheren Verzögerung ist allerdings auch die Gefahr größer, dass bei einer plötzlichen Unterbrechung Daten verloren gehen, wenn der Zwischenspeicher volatil ist. Zusätzlich sind in dieser Zeit die Auswirkungen von Fehlern schlimmer, da die Daten dort nicht weiter gesichert sind.

Durch redundante Pages sind mit *Meta-Cure* (Wang u. a. (2014)) angeblich 70.38% weniger Fehler unkorrigierbar als durch klassische ECC-Strategien. Dabei werden transparent für das Dateisystem Kopien von Pages erstellt, die mit fehlererkennenden Codes ausgestattet sind. Durch die zusätzlichen Zugriffe wird zwar die Lebensdauer insgesamt verringert und die Zugriffszeit steigt, aber die Fehlerwahrscheinlichkeit nimmt ab.

Erweiterter ECC Luo u. a. (2013) stellen ein unter dem Dateisystem liegendes System vor, das gegen gezielte Angriffe besser geschützt ist. Bei zusätzlichen 8 Byte (also insgesamt 36 Byte) pro 4096 Bit können mit einer Chance von $1 - \frac{1}{3*10^8}$ insgesamt 16 unzusammenhängende Bit an Fehlern pro Segment erkannt werden. Dieses Verfahren basiert auf dem erstellten Algorithmus Algebraic Manipulation Detection (AMD) als Ergänzung zu dem *Reed Solomon* Code, der ein sehr verbreitetes Beispiel für ECCs darstellt.

Kalte Redundanz *Kalte Redundanz* bei Datenspeichern bedeutet, dass ein Teil der NAND-Chips im ausgeschalteten Zustand verbleiben, bis einer der im Betrieb befindlichen, kleineren Gruppe von Chips ausfällt. Dann werden die durch normale (*heiße*) Redundanz gewahrten Daten auf den Neuen übertragen, und dieser übernimmt den Platz des Alten. Diese Technik verlängert die MTTF im Gegensatz zu der Idee, alle Speicher gleichzeitig zusammen zu benutzen, deutlich, da ausgeschaltete NAND-Speicher deutlich weniger anfällig für permanente (SEL) Fehler bei gleicher Strahlungsdosis sind (Nguyen u. Scheick, 2003).

Echtzeit FTL Neben den klassischen Ansätzen muss auch beachtet werden, dass in manchen Anwendungsgebieten eine Echtzeitanforderung herrscht. Dabei kann ein geeigneter FTL (Qin u. a., 2012) mit einem RAID-1 (Rao u. a., 2001) verbunden werden, wobei die Verzögerungen der *worst case* Szenarien addiert werden müssen, um die Gesamtvarianz zu bestimmen. Dieser Ansatz hat allerdings die klassischen Nachteile eines *filesystem unaware* FTL, also einem FTL, der nur auf I/O-Ebene Verbesserungen durchführen kann, und keine Kenntnis über z.B. das Schreiben einer großen Datei besitzt. Ein direkt aufgesetztes Dateisystem könnte die Schreibzugriffe und die damit verbundenen Löschvorgänge zusammenfassen, um

die Abnutzung zu verringern.

3.3 Auswertung

Wenn die Echtzeitfähigkeit ein Kriterium ist, ist ein modifiziertes *RTTFS* sinnvoll. *RTTFS* benutzt von sich aus bereits mindestens zwei NAND-Chips, was auf ein Vielfaches von Zwei erweitert werden kann. Dann steigt jedoch nur der Speicherplatz und nicht die Redundanz. Für den Einsatz in einem Strahlungsgebiet ist es allerdings notwendig, eine Fehlererkennung zu implementieren. Die Grundlagen für eine Wiederherstellung sind bereits durch die Redundanz gegeben. Weiterhin ist es für ein solches Einsatzgebiet ausdrücklich sinnvoll, einen *wear leveling* Algorithmus hinzuzufügen. Für eine besonders lange Betriebsdauer ist es zusätzlich noch empfehlenswert, einen Mechanismus hinzuzufügen, der die Anzahl redundanter Kopien erhöht. Das Hinzufügen kalter Redundanz ist allerdings eine sehr große Herausforderung für Echtzeitsysteme, da das Kopieren eines ganzen NAND-Chips sehr viel Zeit in Anspruch nimmt, und somit entweder die Latenzanforderungen heruntergesetzt oder ein weiterer, vergleichsweise komplexer, Zwischenspeicher eingerichtet werden müssen.

Eine arbeitsintensive Variante ist, *RCFFS* zu erweitern. Der Vorteil dieses Dateisystems ist, dass in höheren Schichten erkannt werden kann, ob der NAND-Chip langsam altert, um dann die Anzahl redundanter Pages und Blöcke auf Kosten der Speicherkapazität zu erhöhen. Das macht das Dateisystem sehr flexibel. Es sollte allerdings der im Paper beschriebene *garbage collector* mit einem *consistency checker* implementiert werden, weiterhin sind *wear leveling* und redundante Auslagerungen auf weitere Chips sowie Ersatzchips (*kalte Redundanz*) essenziell für eine längere Mission im Weltraum. Daher sollten diese Module noch hinzugefügt werden.

Weniger arbeitsintensiv wäre es, einen normalen RAID-1 Modus mit einem FTL zu verbinden. Dies bringt zwar die bekannten Nachteile eines *filesystem unaware* FTL mit sich, ist aber besser zu testen, auszutauschen und zu implementieren, da die Komponenten separat lauffähig sind und damit ihre eigenen Abstraktionen mit sich führen. Der FTL kann hierbei auf Echtzeitfähigkeit ausgelegt werden (Qin u. a., 2012), allerdings ist eine Abwägung zwischen der Notwendigkeit von verlässlichen Antwortzeiten und einer höheren Zugriffsgeschwindigkeit notwendig, da solche Systeme wegen vergleichsweise seltenen, aber langen Antwortzeiten die Schnelleren verzögern, um ein vorausbestimmbares Verhalten zu produzieren. Bei der Wahl einer angemessenen RAID-1 Implementation ist darauf zu achten, dass sie von sich aus regelmäßig *sweeping* vornimmt.

3.4 Integrierte Dateisysteme

Für diese Bachelorarbeit wurden *FAT* und *YAFFS-1* ausgewählt. *FAT* wurde gewählt, weil es ein sehr einfaches, aber bekanntes Dateisystem ist. Es bietet einen

einfachen Einstieg in die Programmierung des Treibers und dient als Beispiel für ein nicht auf NAND-Flash ausgelegtes Dateisystem. Dadurch können aussagekräftigere Vergleiche erzielt werden. Die Wahl des zweiten Dateisystems fiel auf *YAFFS-1*, da es auch sehr weit verbreitet ist, der Treiber durch das *yaffs-direct* Interface ohne viele externe Abhängigkeiten implementiert werden kann, und es die grundlegenden Sicherungsmechanismen unterstützt. *YAFFS-2* wurde nicht gewählt, weil *YAFFS-2* keinen eigenen ECC mitliefert, und es keine weiteren Vorteile gegenüber der simulierten Speicherkonfiguration bietet[5]. Weitere Dateisysteme konnten aufgrund der Zeitbeschränkung der Bachelorarbeit nicht getestet werden.

FAT steht für *File Allocation Table* und ist ein sehr einfaches, weit verbreitetes Dateisystem. Die verwendete Implementation des Dateisystems ist von Elm Chan[6] und ist unter einer *Open Source* Lizenz veröffentlicht worden. *FAT* hält in Anfangssektoren nach einem Bootsektor und Informationen über das Dateisystem eine *file allocation table* (oder auch Master File Table (MFT)), in der Positionen der Dateifragmente abgelegt werden. Ist die Nummer des ersten Fragments (oder *cluster*) bekannt, wird dort nachgeschlagen, ob es ein weiteres Fragment in einer Kette gibt, oder dieses das Ende einer Datei oder eines Verzeichnisses darstellt. Oft wird eine Kopie der MFT unterhalten, um bei Schreibfehlern oder Ausfällen noch eine Chance zu haben, Daten zu retten. Dies ist allerdings nur unter günstigen Umständen möglich und ist im Standard nicht vorgesehen.

YAFFS-1 *YAFFS-1* unterscheidet sich zu seinem Nachfolger *YAFFS-2* im Wesentlichen durch die Annahme, dass eine Page zwei bis drei Mal beschrieben werden kann. Allerdings produzieren neuere Flashspeicher dadurch undefiniertes Verhalten[7]. Zusätzlich wurden in der ersten Version nur Speicher mit 512 Byte breiten Pages unterstützt. Der *YAFFS-1* eigene ECC kann pro 256 Byte ein Bit korrigieren und zwei Bitfehler erkennen. Der Nutzen von *write twice*, also dem zweimaligen Beschreiben einer Page, ist, dass eine veraltete Page als solche markiert werden kann, ohne wie *YAFFS-2* eine aufsteigende Seriennummer unterhalten zu müssen. Wird ein Datenbereich logisch verändert, werden die korrespondierenden Pages durch den zweiten Zugriff als *dirty* markiert, und die nächsten freien Pages werden mit den neuen Daten beschrieben. Enthält ein Block ausschließlich veraltete Pages, wird dieser gelöscht. Wenn es nur noch wenige freie Pages gibt, wird die *garbage collection* eingeleitet. Diese sucht Blöcke mit möglichst vielen veralteten Pages, und verschiebt die Verbliebenen, noch Benutzten in einen freien Bereich. Anschließend kann der Block gelöscht werden. Diese Vorgehensweise bewirkt ein dynamisches *wear leveling*, da häufig veränderte Pages sukzessive über alle freien Positionen verteilt werden, nicht jedoch über unveränderte, aber gültige Datensätze. Die Positionen von Objekten (wie Dateien, Ordner, Links u.A.) stehen nicht in einer Tabelle, sondern müssen

[5] Der simulierte NAND-Speicher kann mehrmals beschrieben werden, und hat die richtige Pagegröße. (siehe Abschnitt 4.2

[6] http://elm-chan.org/fsw/ff/00index_e.html

[7] http://www.yaffs.net/documents/how-yaffs-works

durch einen Suchlauf über den kompletten Speicher bei dem *mounting* gefunden und im Arbeitsspeicher verlinkt werden. Das beeinflusst zwar die Zeit für das *mounting* sehr (linearer Aufwand in Abhängigkeit der Speichergröße), birgt aber auch Vorteile bezüglich der Lebensdauer (vgl. *CFFS*, Abschnitt 3.2.1). Die Zugehörigkeit und Abfolge von Pages einer Datei wird durch eindeutige Dateinummern gewährleistet.

Kapitel 4

Die Simulationsumgebung

Die Simulationsumgebung ist dazu gedacht, Dateisysteme im Bezug auf ihre Zuverlässigkeit in Raumfahrtumgebungen zu evaluieren. Besonders von Interesse ist daher, wie ressourcenschonend ein Dateisystem mit dem Speicher umgeht, und wie robust es auf Fehler wie z.B. Bitkipper reagiert. Da die Performanz im Sinne der Bearbeitungsgeschwindigkeit in vielen Raumfahrtanwendungen in der Wichtigkeit hinter Langlebigkeit steht, wird dieser Aspekt nicht betrachtet. Falls es von Interesse ist, können existierende Simulatoren wie z.B. *DiskSim* (siehe Abschnitt 1.3) benutzt werden. Auf eine Simulation des Energieverbrauchs oder z.B. der Temperaturverteilung ist verzichtet worden, da diese nicht durch ein Dateisystem überwacht oder geregelt werden.

4.1 Übersicht

Die Simulationsumgebung besteht aus drei Abstraktionsebenen. Ganz oben befindet sich die Testebene, auf der unabhängig von den darunterliegenden Dateisystemen Anwendungsszenarien simuliert werden. Diese Abstraktion erlaubt, dass Ergebnisse von getesteten Dateisystemen vergleichbar werden.
In der Mitte liegt die Dateisystemschicht, die für jedes Dateisystem das Minimum der POSIX-Funktionen für die Tests bereitstellen muss, die eigentliche Logik der Dateisysteme enthält und jeweils Treiber für die Interaktion mit dem Flash beinhaltet. Weiterhin ist hier die Abstraktion von Fehlerursachen und deren Auswirkungen auf den Flash für die Benutzung in Tests eingereiht.
Zuletzt gibt es die Simulationsebene, die mit der Schnittstelle eines üblichen Flashcontrollers eine NAND-Flash Zelle simuliert, und die durch ein erweitertes, zweites Interface die Möglichkeit der Fehlerinjektion bietet.
Diese Ebenen sind so gewählt worden, dass verschiedene Dateisysteme transparent gewechselt werden können, ohne Tests oder die NAND-Simulation zu verändern.

Benutzung Soll ein neues Dateisystem getestet werden, muss es zunächst zwei Schnittstellen implementieren. Zum Einen ist dies ein NAND-Treiber, um mit der NAND-Simulation (die Klasse *FlashCell*) zu interagieren. Dieser implemen-

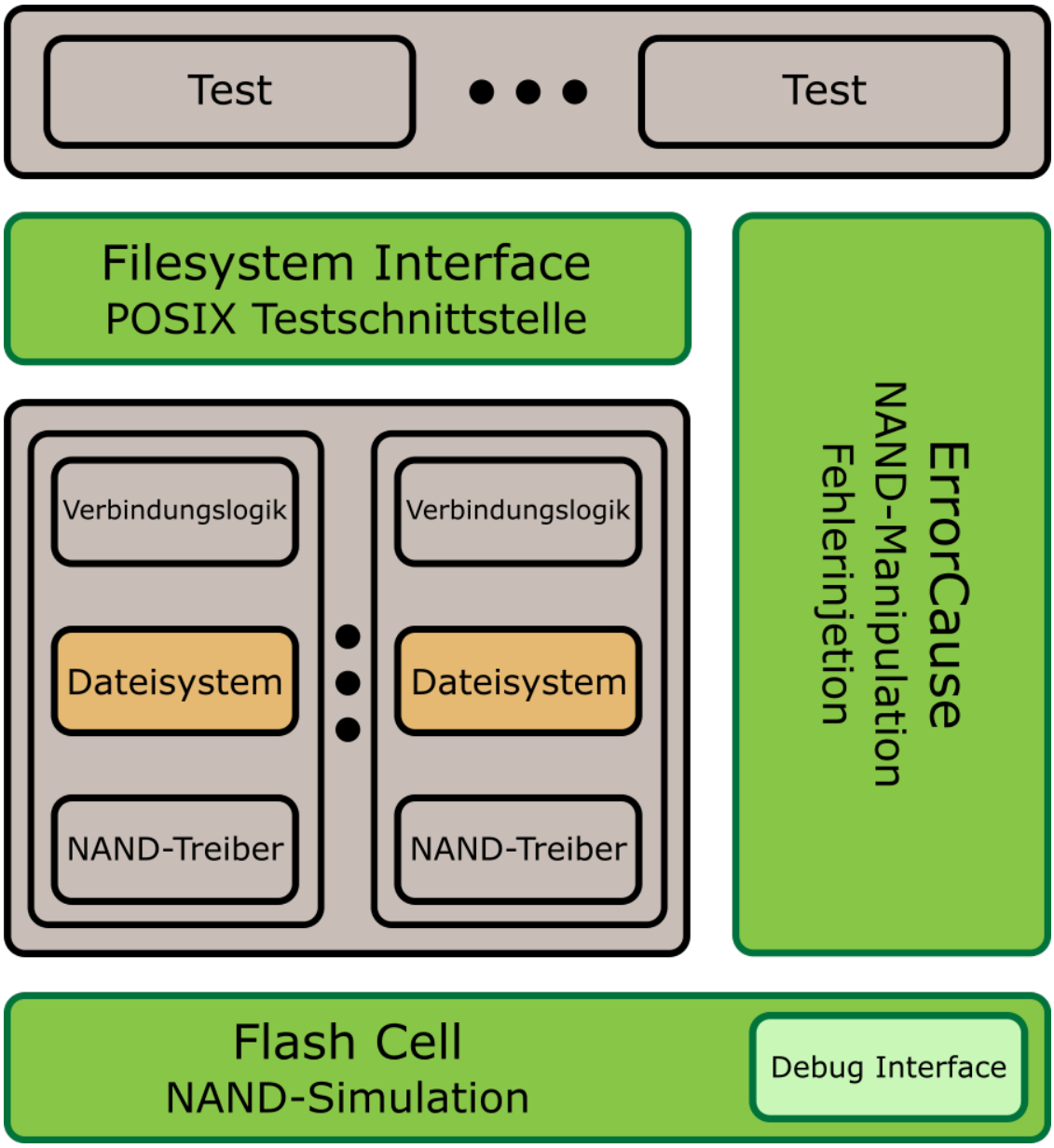


Abbildung 4.1: Aufbau der Komponenten der Testumgebung und die Position der benutzerdefinierten Modulen.

tiert die grundlegenden Interaktionen mit der Simulation, damit das Dateisystem darauf zugreifen kann. Zum Anderen muss eine Verbindungslogik zu der POSIX-Testschnittstelle erstellt werden, die Dateisystemoperationen wie z.b. *f_open* abstrahiert. Durch sie können eventuelle Eigenheiten des Dateisystems, wie z.B. abweichende Datentypen für einen Dateideskriptor, transparent für den Test angepasst werden. Sind diese Schnittstellen vorhanden, kann das Dateisystem mit den vorhandenen Tests getestet werden.
Soll ein neuer Test entwickelt werden, muss er nur mit der POSIX Testschnittstelle und den NAND-Manipulationen interagieren. Ohne etwas an Dateisystem oder dem NAND-Flash ändern zu müssen, können neue Fehlerklassen mithilfe der Klasse *ErrorCause* erstellt werden. Diese wird für die Injektion von Fehlern benutzt, um unabhängig von dem Test definierte Fehlerquellen anzubieten.

4.2 Modell NAND

Das Modell des NAND-Speichers beschränkt sich auf Fehlerquellen und deren Auswirkungen, die durch Software lösbar sind (siehe Tabelle 2.1). Nicht betrachtet wird ein erhöhter Stromverbrauch und Verzögerungen in der Ausführung von Operationen. Weiterhin sind Ursachen wie der Ausfall der Ladungspumpe und der Thyristor-SEL ausgelassen, weil diese nicht durch ein Dateisystem oder verwandte Software unterdrückt werden können. Zur Reduzierung der Komplexität sind Eigenschaften des Speichercontrollers wie *state machine*, abwechselnde Verteilung von Blocknummern unter den Planes und Zwischenregister für jede Page (vgl. Abb. 2.1) nicht implementiert, da keine Performanz simuliert werden muss. Die möglichen Fehlerursachen aus *statemachine* und Pageregister sind direkt in der Hauptklasse *FlashCell* implementiert und werden von einer entsprechenden Implementation von *ErrorCause* durch das *DebugInterface* ausgelöst.

Das *DebugInterface* ist ein Teil der *FlashCell*, da es Aktionen erlaubt, die sonst nicht über ein NAND-Flash Interface erreichbar sind. Dazu gehört z.B. das Auslesen einzelner Bytes, ohne dass die Abnutzungswerte (*ACCESS_VALUES*) verändert werden. Das *DebugInterface* wird nicht vom Dateisystem benutzt, sondern ist nur dazu da, im laufenden Betrieb Fehler zu injizieren oder Inhalte an eine (hier nicht behandelte) grafische Oberfläche zu senden. Nach dem Betrieb können mit einem *FcSerializer* durch das *DebugInterface* die Inhalte der *FlashCell*-Klasse serialisiert werden, um detaillierte Ergebnisse zu sichern und Statistiken zu erstellen.

Benutzung Die NAND-Simulation besteht aus verschiedenen Klassen, die der Struktur eines NAND-Bausteines (Plane, Block, Page, Byte) nachempfunden sind. So können Funktionen besser gekapselt werden. Weiterhin sollen *structs* die Verwendung vereinfachen. Die Struktur *FAILPARAM* zum Beispiel wird bei der Erstellung einer *FlashCell* benutzt, um die Ausfallparameter zu definieren. Sie fasst die Parameter zusammen, mit denen für jedes *F_byte* einzeln die Ausfallparameter (*FAILPOINT*s) berechnet werden. Die Berechnung sieht eine Gaußsche Zufallsver-

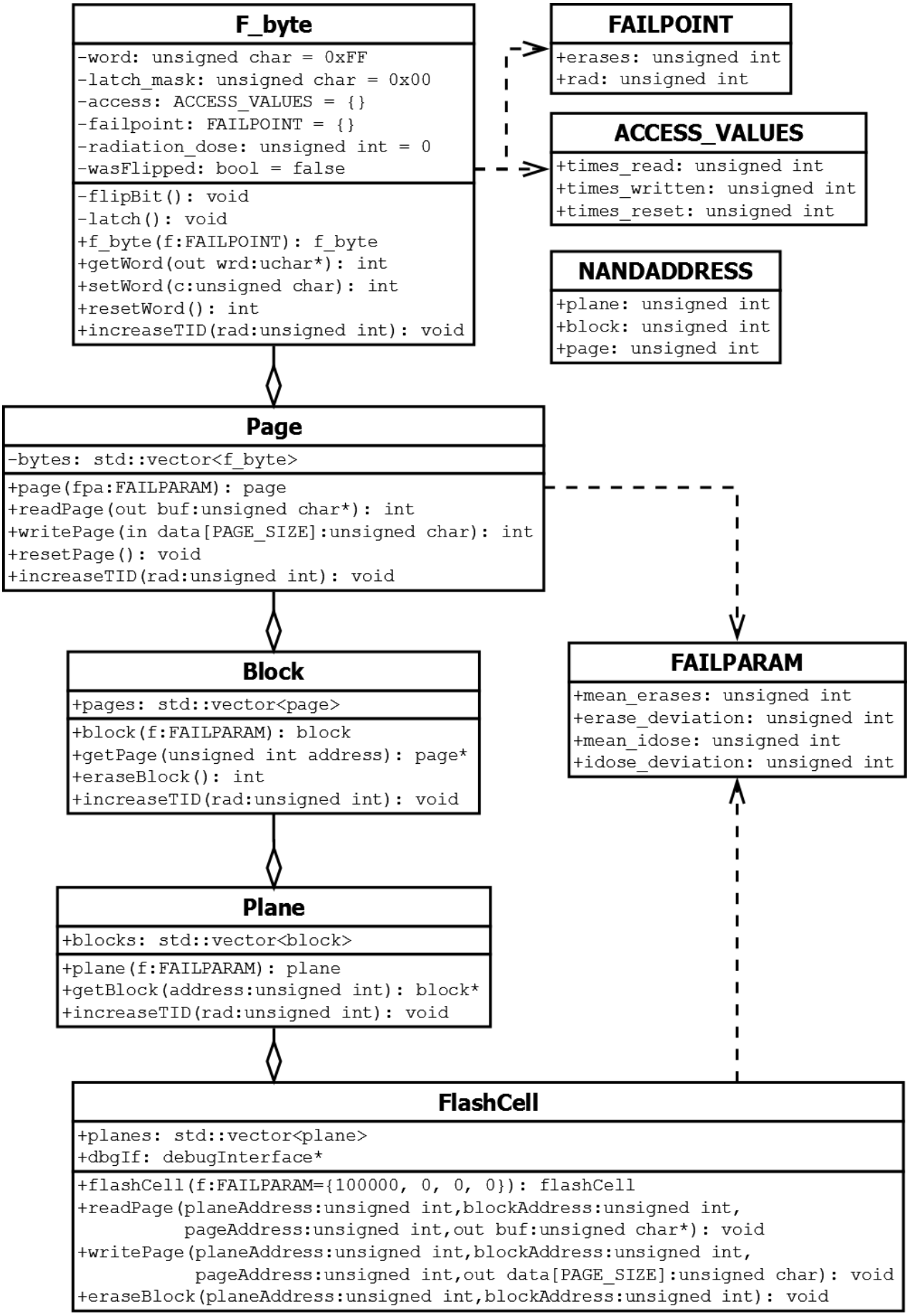


Abbildung 4.2: Klassendiagramm des simulierten Flashspeichers. Es wird das typische Layout von Flashspeichern nachgebildet.

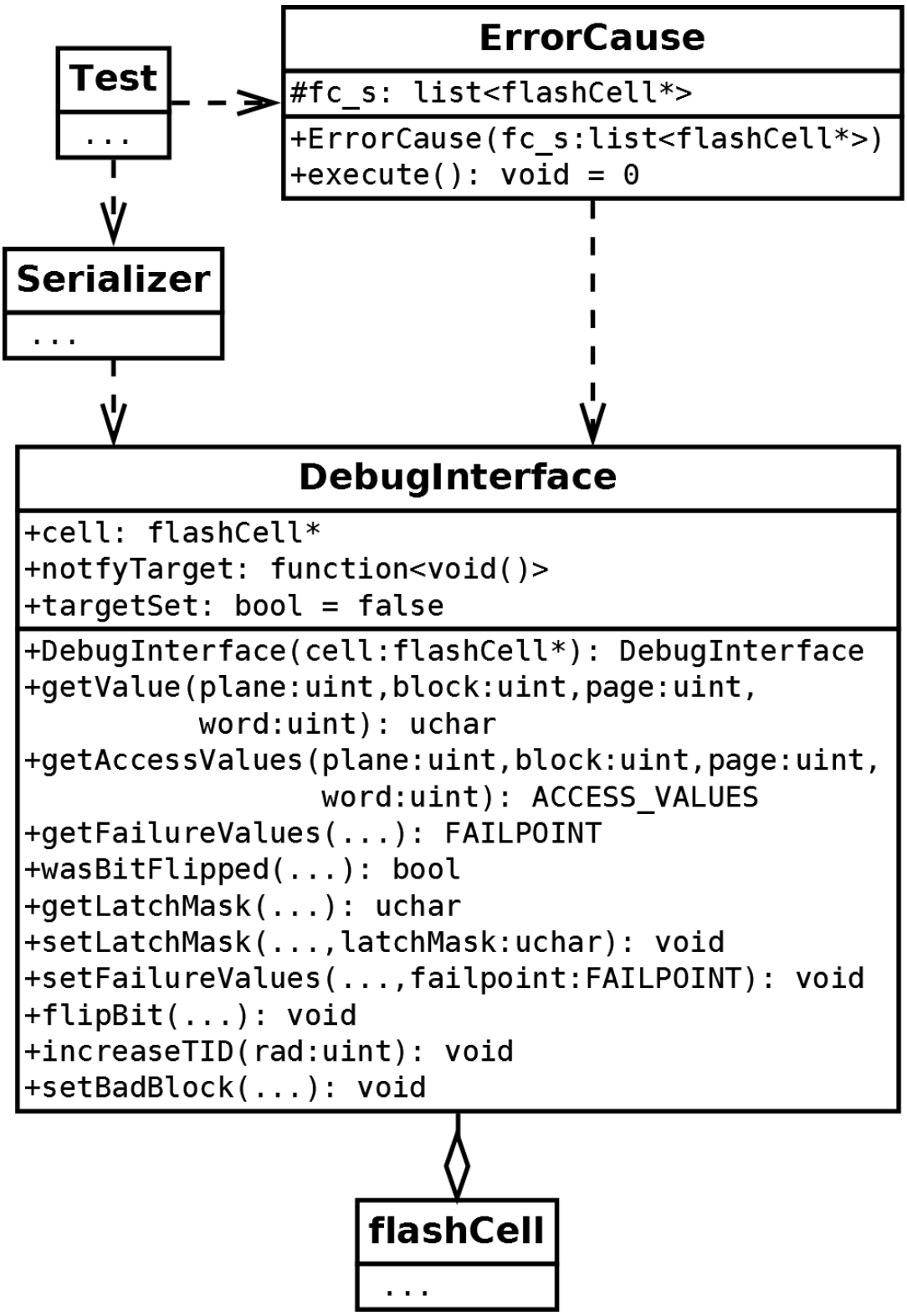


Abbildung 4.3: Die Beziehungen des Debug Interfaces.

teilung mit einem Mittelwert *mean_erases u. -idose* und einer Varianz *erase- u. idose_deviation* vor. Die Struktur *FAILPOINT* beinhaltet die Information, nach wie vielen Löschvorgängen oder wie viel Strahlung ein konkretes Byte ausfällt. Die Anzahl der Zugriffe und die aktuelle Strahlungsdosis werden von jedem *F_byte* in der Struktur *ACCESS_VALUES* gespeichert.

Die in den Tests verwendete NAND-Struktur ist frei wählbar und entspricht einer 528 KiB großen Flasheinheit inkl. OOBA, bestehend aus einer *logical unit*, die 4 Planes á 4 Blöcke enthält. Pro Block gibt es 64 Pages, die jeweils 528 Byte breit sind (512 Byte Nutzdaten + 16 Byte OOBA). Die Struktur ist so gewählt worden, dass der Speicher nicht zu groß ist, aber immer noch ein gängiges Format besitzt. Die komplette Simulationsumgebung belegt mit dieser Einstellung ungefähr 16 MiB Arbeitsspeicher.

Im Schnitt weist die in den Tests verwendete Ausfallkonfiguration *[Mittlere TID: 30 kRad, Abweichung: 5000 Rad]* nach einer TID von ungefähr 8 kRad 5 *latchups* und 15 Bitkipper auf. Die ersten Bitfehler treten bei NAND-Flash meistens zwischen 8-14 kRad auf (Nguyen u. a. (1998), Nguyen u. Scheick (2003)). Die Parametrisierung kann an den verwendeten Flashspeicher angepasst werden. Z.B. entspricht die Konfiguration *[Mittlere TID: 48 kRad, Abweichung: 5500 Rad]* ungefähr dem in Nguyen u. Scheick (2003) geschilderten „Intel 256 MiB MLC" Flashspeicher. Die Abnutzung (*erase cycles*) wurde in diesem Beispiel allerdings nicht betrachtet.

Um einen speziellen Speicher detailgetreu abbilden zu können, müssen die Eigenschaften des realen Speichers bekannt sein. Das kann z.B. durch Auslastungs- bzw. Strahlentests mehrerer Instanzen des zu testenden Speichers erfolgen. Daraus können dann nach einer statistischen Auswertung die Ausfallparameter errechnet werden.

4.3 Fehlerklassen

Die abstrakte Fehlerklasse *ErrorCause* bietet eine Möglichkeit, NAND-Flash Manipulationen strukturiert anzuwenden. Es wird eine Reihe von Fehlerursachen (vgl. Tabelle 2.1) angeboten, die, durch Parameter anpassbar, von den Tests benutzt werden können. Zum Beispiel wird die Klasse *Cause::IncreaseTID* von dem Test *Test_fileIO_TID* benutzt.

4.4 Treiber

Die Treiber ermöglichen den Dateisystemen, mit der programmierten NAND-Simulation zu interagieren. Meistens verlangt das Dateisystem eine Anzahl von Funktionen von dem Treiber, die sich nach einer Spezifikation verhalten sollen. Wenn das Dateisystem z.B. von einem block device ausgeht, können somit nicht alle Operationen atomar ausgeführt werden, was zu einer erhöhten Abnutzung des Flashes führt. Zum Beispiel muss bei FAT eine Schreiboperation für einen einzigen

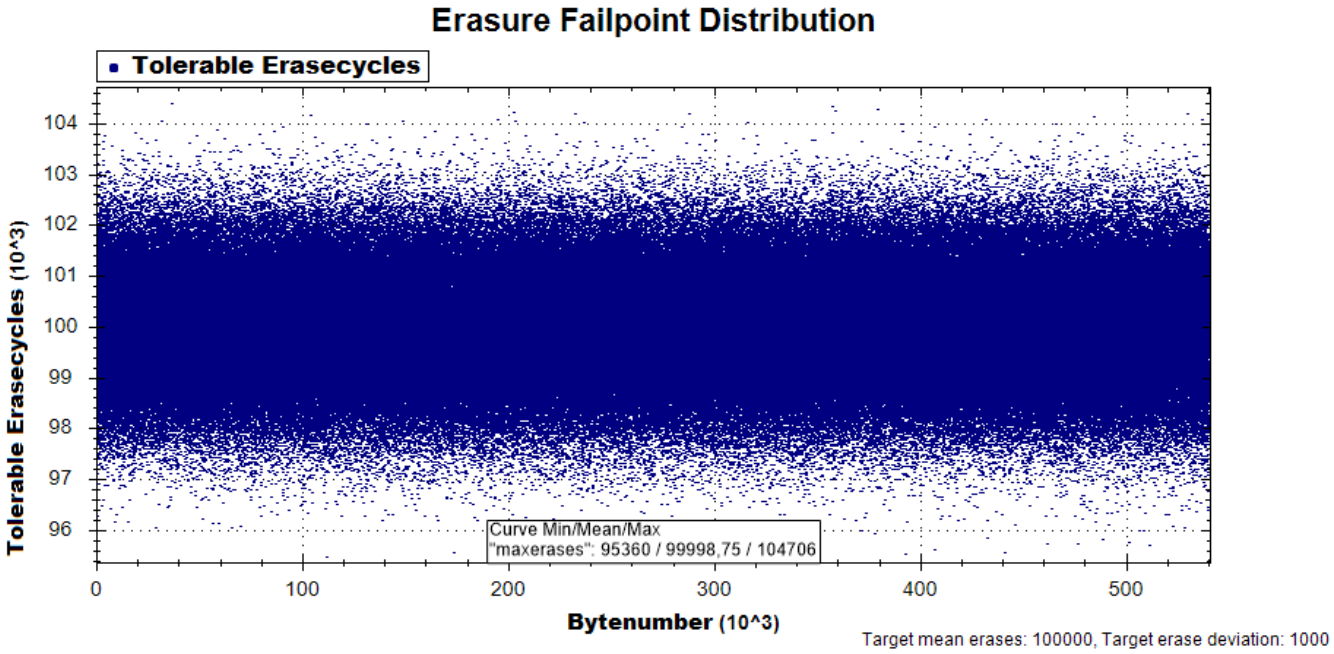


Abbildung 4.4: Die Verteilung der *FAILPOINT*s aller Bytes einer Instanz von *FlashCell*. Betrachtet werden nur die Löschzyklen, die Strahlungsdosis wird analog errechnet. Zu erkennen ist die Normalverteilung um die Sollwerte: Mittelwert: 100.000, Standardabweichung: 1000.

FAT Block (eine NAND-Page) implementiert werden. Daher wird der Treiber zuerst den gesamten NAND-Block zwischenspeichern, löschen, den Zwischenspeicher modifizieren, und wieder zurückschreiben. Da alle verwendeten Dateisystemimplementationen (für *FAT ff11* und für *YAFFS-1 yaffs1*) in der Programmiersprache *C* geschrieben sind, müssen die Treiber zusätzlich eine Übersetzung von der objektorientierten NAND-Simulation in *C++* nach *C* beinhalten. Beide Treiber benutzen nur eine Instanz von *FlashCell*, da die Dateisysteme keine RAID-ähnlichen Techniken unterstützen, und somit nur den Zugriff auf ein Gerät gleichzeitig ermöglichen.

4.5 Dateisysteminterface

Das Dateisysteminterface (*FsIF*) ermöglicht, dass ein Test unabhängig des tatsächlichen Funktionsumfanges eines Dateisystems von einer festgelegten Menge an Funktionalität ausgehen kann. Es liefert Dateifunktionen wie *f_open* oder *f_stat*, kann aber auch Dateisysteminformationen wie z.B. den freien Speicherplatz oder den letzten Fehler anzeigen.

Benutzung Das *FsIF* muss für jedes Dateisystem implementiert werden. Um eine allgemeine Dateischnittstelle zu bieten, muss das in Abschnitt 2.3 beschriebene Mindestmaß an I/O Funktionen angeboten, und in das dateisystemspezifische Layout übersetzt werden. Da POSIX ein weit verbreiteter Standard ist, wurde das Layout daran angepasst. Wenn alle Funktionen mit den selben Parametern bereits

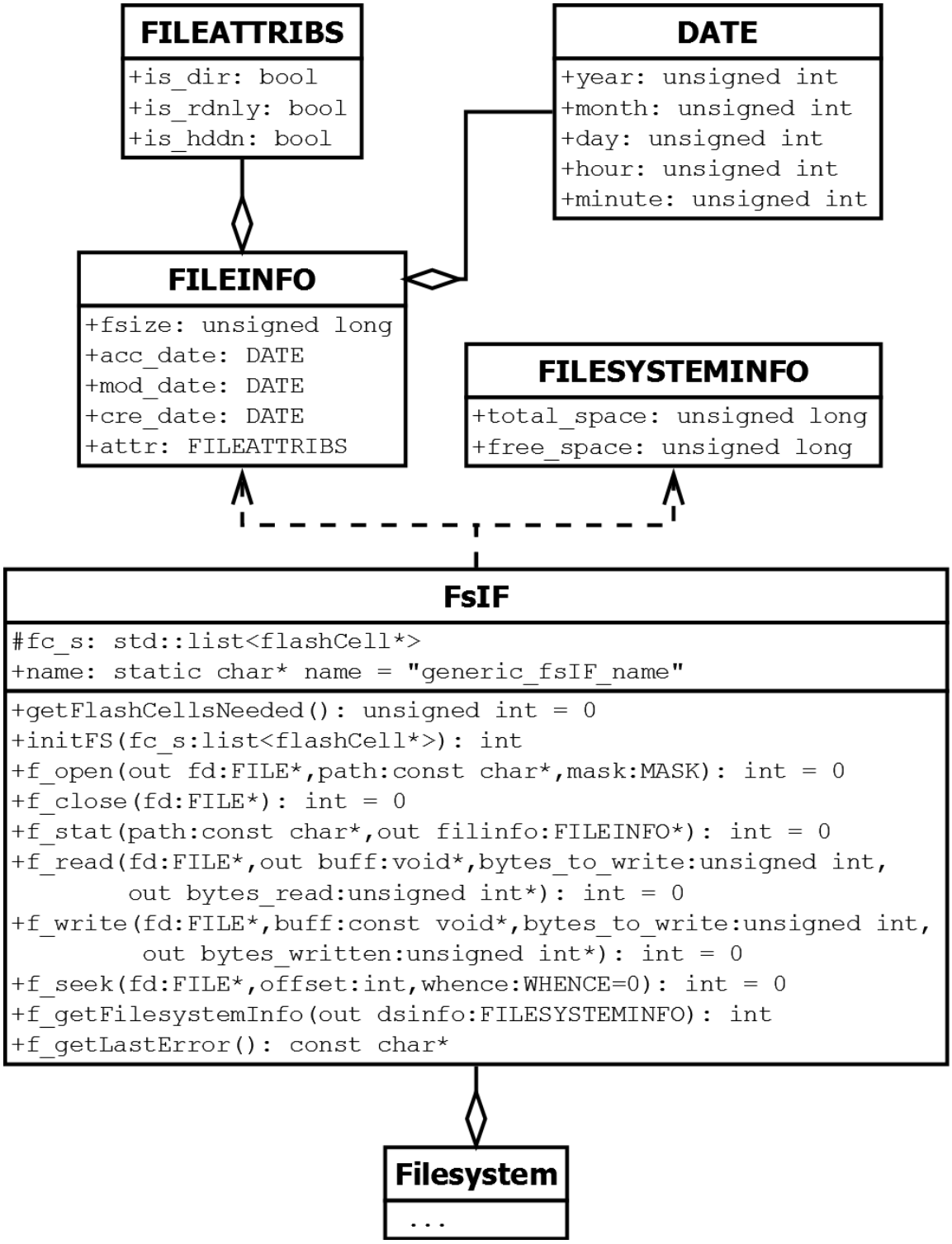


Abbildung 4.5: Diagramm der abstrakten Dateisysteminterfaceklasse.

vorhanden sind, ist die Übersetzung transparent; es können auch grundlegende Unterschiede sein, wie z.B. bei *ff11*, der in diesem Projekt benutzen Implementation von FAT. Dort sind Dateideskriptoren nicht, wie in POSIX üblich, Nummern, sondern *structs*, die einen Dateikontext enthalten. Daher muss die Implementierung von *FsIF* für *ff11* eine Liste geöffneter Dateien unterhalten, um bidirektional übersetzen zu können.

4.6 Tests

Die Klasse *Test* kann auf alle Funktionen der POSIX-Testschnittstelle und der NAND-Manipulationsklassen *ErrorCause* zugreifen, sowie eigene Funktionen implementieren. Sie simuliert die Auslastungsprofile, die für die spätere Anwendung erwartet werden. Die zu testende Anwendung kann auch direkt als Testklasse in SATFON laufen, um das Profil so naturgetreu wie möglich nachzustellen. Dafür muss sie zusätzlich von der abstrakten Testklasse erben, und die notwendigen Funktionen implementieren.

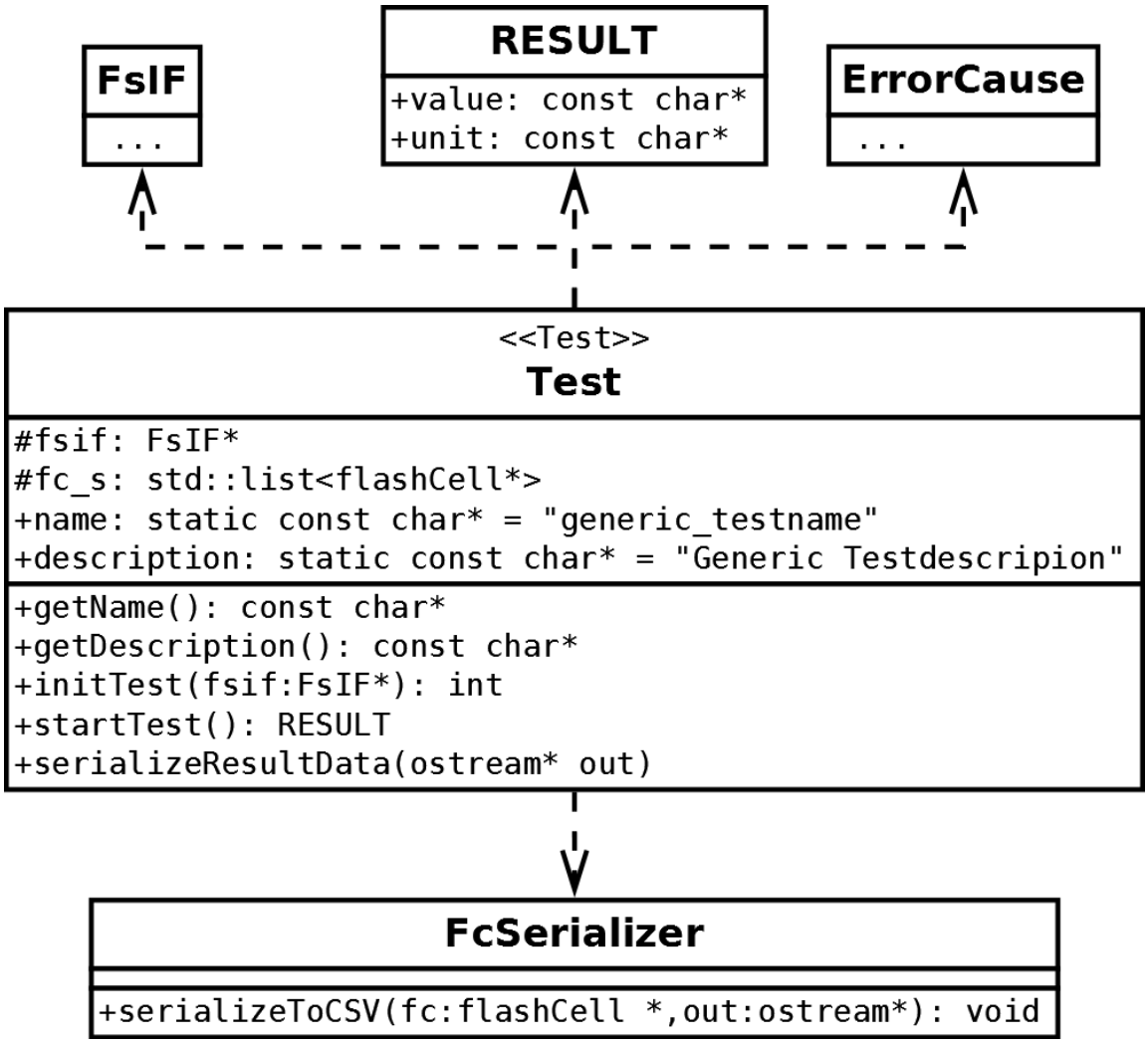


Abbildung 4.6: Überblick der von Tests benutzten Klassen.

Benutzung Eine Implementation der Klasse *Test* sollte reproduzierbare Anwendungsfälle simulieren und ein auswertbares Ergebnis produzieren. Vor dem Start der eigentlichen Testroutine können Vorbedingungen geschaffen werden, wie z.B. das Einfügen von fehlerhaften Pages oder das Formatieren des Dateisystems. In der Haupttestroutine kann z.B. eine Datei gelöscht und beschrieben werden, bis ein Fehler auftritt (*Test_fileIO*). Der Test selber bestimmt, welche Serialisierungsmethode sinnvoll ist, um das gefundene Ergebnis zu visualisieren. Wenn die Funktion *serializeResultData(...)* aufgerufen wird, wird die Anfrage direkt an eine Implementation von *FcSerializer* weitergeleitet, und die Daten in eine Datei geschrieben.
Eine Auflistung der implementierten Tests findet sich in Abschnitt 5.1.2.

4.7 Sonstige

Serialisierung Die abstrakte Serialisierungsklasse *FcSerializer* bietet die Möglichkeit, nach Beendigung eines Tests die relevanten Aspekte der NAND-Zellen in ein CSV Format zu überführen. Es werden verschiedene Aspekte in Implementationen der Klasse angeboten, die jeweils von den Tests ausgesucht werden. Zum Beispiel gibt die Klasse *BitflipLatchupPageSerializer* eine Tabelle aus, die für jede Page die Anzahl der insgesamt aufgetretenen Bitkipper und Latchups anzeigt. Die Klasse *PageSerializer* erstellt analog dazu eine Tabelle mit den *ACCESS_VALUES* *erases* und *writes*.

Verwaltung von Tests und Dateisystemen Da ein Benutzer einfach alle verfügbaren Tests und Dateisysteme anzeigen und auswählen können soll, wurde die Klasse *InstanceRegistry* implementiert. Sie ermöglicht es, Klassen zu instantiieren, die nur durch einen Textnamen bekannt sind. Die Konstruktoren der Klassen werden mit ihrem Namen und einer Beschreibung statisch durch Headerdateien registriert. Im Moment werden in dem Kommandozeilenprogramm von SATFON alle verfügbaren Tests und Dateisysteme angezeigt, wenn ungültige oder zu wenig Parameter angegeben werden. Will ein Benutzer einen Test und ein Dateisystem auswählen, wird der gegebene Name des Tests bzw. Dateisystems in der entsprechenden Instanz von *InstanceRegistry* in einer Liste gesucht. Wird die Klasse gefunden, gibt der Aufruf eine neue Instanz zurück.

Erzeugung von Statistiken Da manche Tests (*fileIO_SEU*, *fileIO_SEL*, *fileIO_TID*) stark von zufälligen Ereignissen abhängen, werden diese Tests mit einem Skript mehrmals ausgeführt, und das Ergebnis jeweils in eine Datei angehängt. Durch ein selbstgeschriebenes Programm (*ergebissortierung.c*) wird die Datei eingelesen, die Werte in „Wert-Vorkommnisse"-Paaren sortiert, und die Paare formatiert ausgegeben. Diese Daten können dann durch einen CSV-Plotter visualisiert werden. Weiterhin wurde die Ermittlung des Mittelwertes und des Medians sowie des Maximal- und Minimalwertes durch *ergebnisstatistik.c* automatisiert, welches die Ausgabedatei des Skriptes einliest und die Statistiken errechnet.

4.8 Beispielabläufe

Wird in einem Test eine Datei gelesen, ruft er von der gegebenen *YAFFS*-Implementation der Testschnittstelle *FsIF f_read()* auf. Diese übersetzt die Funktion zu *yaffs_read()* (andere Dateisysteme analog). Das Dateisystem stößt nun, je nach Lesegröße, eine Reihe von Leseoperationen an, um die korrespondierenden Teile (*chunks*) der Datei zu sammeln. Die Funktion, *chunks* auszulesen, ist im Treiber implementiert, und liest eine Page aus der *FlashCell* aus. Diese Teile werden von dem Dateisystem wieder zusammengesetzt, und an den Test zurückgegeben.

Das Sequenzdiagramm in Abb. 4.8 beschreibt den Vorgang, wie NAND-Manipulationen benutzt werden können. Wenn ein Test (wie die Klasse *test_fileIO_SEU*, Abschnitt 5.1.2) SEUs injiziert, wird zuerst in der Initialisierungsfunktion des Tests die Implementation von *ErrorCause induce_SEU* konfiguriert. Später, in der Testroutine, wird nur noch *execute()* zwischen dem Schreiben und anschließendem Verifizieren aufgerufen. Diese Funktion bewirkt, dass die Klasse *induce_SEU* zufällige Adressen generiert, und das *DebugInterface* aufruft. Dieses wird an diesen Stellen ein Bit kippen, und, falls angewiesen, die TID erhöhen. Da das *DebugInterface* vollen Zugriff auf die internen Variablen der zugehörigen *FlashCell* besitzt, führt es die Anfrage ohne Umwege aus.

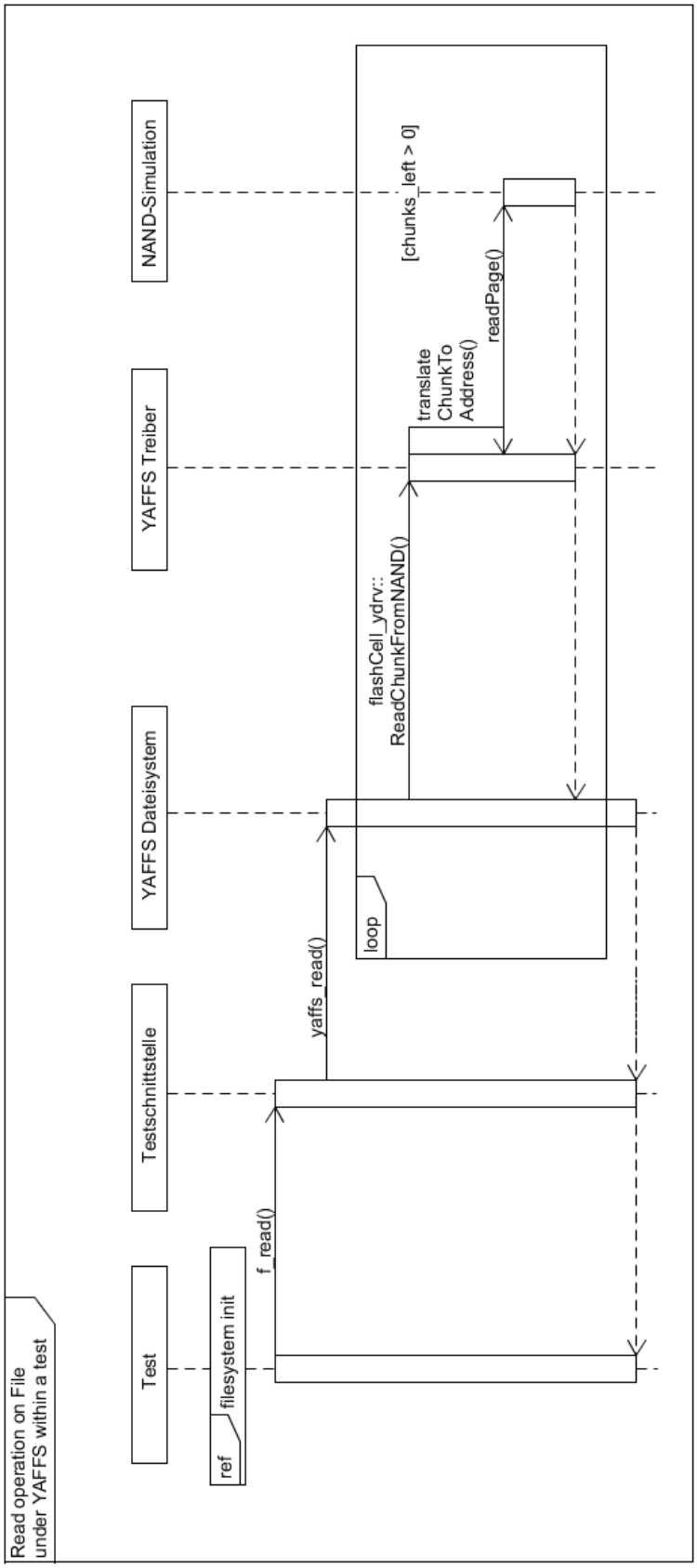


Abbildung 4.7: Sequenzdiagramm eines Lesevorganges unter *YAFFS* (*FAT* vergleichbar)

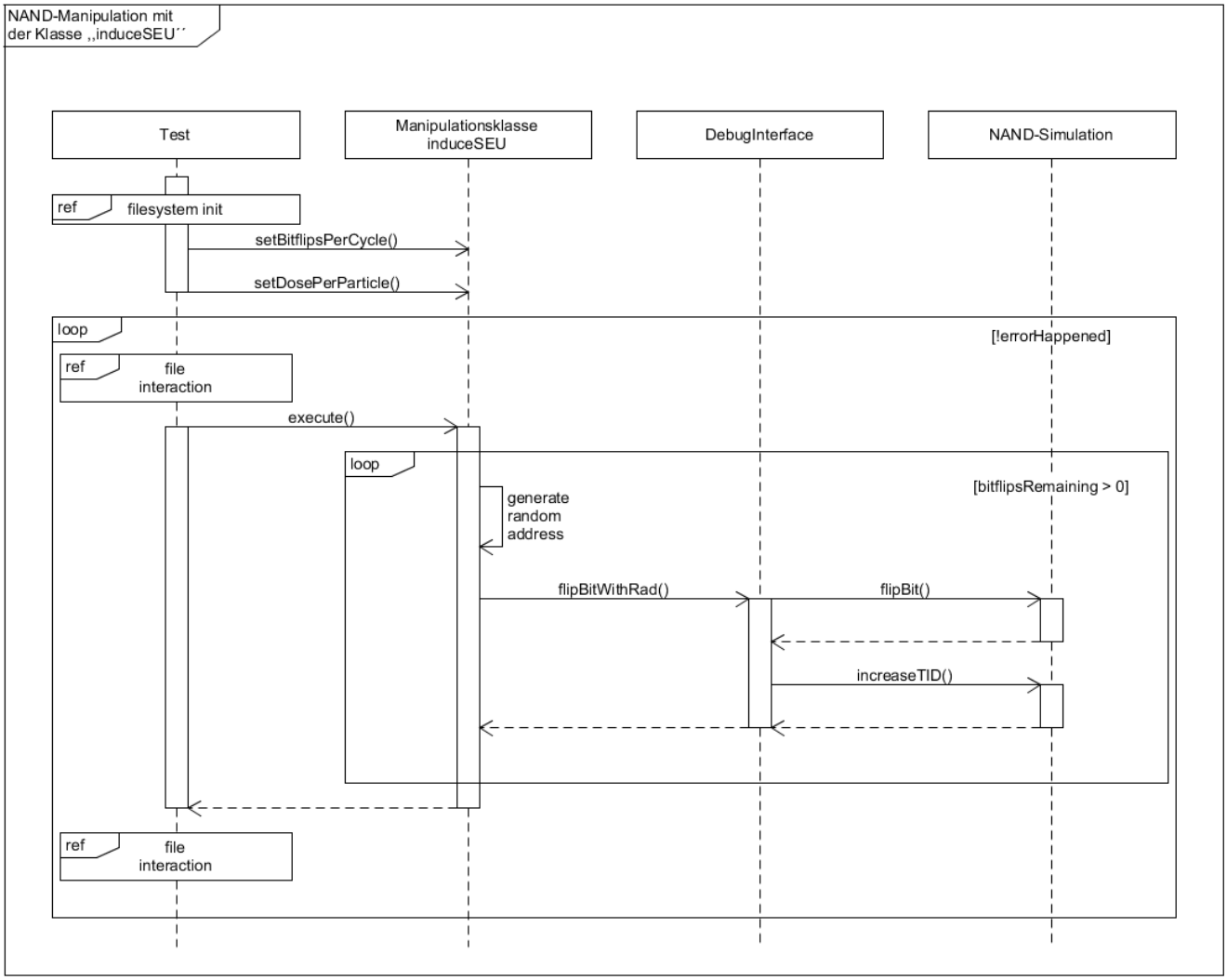


Abbildung 4.8: Sequenzdiagramm der Injektion von *single event upset*s durch einen Test.

Kapitel 5

Auswertung

5.1 Grundlagen

5.1.1 Implementierte Fehlerquellen

Um alle Fehlerquellen abzudecken, wurden vier Implementationen von *ErrorCause* erstellt, die von Tests frei kombinierbar sind.

induce_SEU Die Klasse *induce_SEU* soll einzelne Bit an gegebenen Positionen kippen, und falls konfiguriert, die dortige TID erhöhen. Sie stellt SEUs nach, die aufgrund von Partikeleinschlägen entstehen.

induce_SEL Die Klasse *induce_SEL* bringt an zufälligen Positionen Bits zum *latchup*. Sie stellt SELs nach, die aufgrund von Partikeleinschlägen entstehen.

increase_TID Diese Klasse stellt Fehler aufgrund von einer erhöhten Total Ionizing Dose nach, und kann daher die TID aller *FlashCell*s erhöhen. Der Ausfall selbst wird in der Klasse *FlashCell* durchgeführt, da jedes Byte eine andere Strahlendosis aushalten kann (siehe Abschnitt 4.2). Bei Näherung an die maximale Strahlungsdosis beginnen Bytes, Bitkipper zu produzieren, bei Überschreitung fallen die Bytes aus, indem sie zufällige Werte produzieren. Eine Lese-, Schreib- oder Löschaktion gibt zusätzlich einen Fehler zurück, der dem Treiber signalisiert, dass der Vorgang ungültig war.

fabricationDefect Diese Klasse soll den Fehlerfall abdecken, dass Blöcke bereits seit der Auslieferung defekt sind, da eine *bitline* (siehe Abb. 2.3) nicht leitet, und somit alle Pages an der selben Stelle ein nicht programmierbares Bit besitzen. Da in der Realität der Hersteller alle als nicht funktionstüchtig erkannten Blöcke als solche markiert, wird auch hier ein *bad block marker* gesetzt. Dazu wird bei 8bit SLC Flash das sechste Byte der OOBA mit einer Zahl ungleich *255* beschrieben.

Der Ausfall aufgrund erhöhter TID und die Abnutzungserscheinungen des

Speichers werden in der *FlashCell* ohne *ErrorCause* implementiert, da dies direkte Reaktionen des Flashes sind.

5.1.2 Implementierte Tests

Die Aussagekraft der erstellten Statistiken hängt stark von der Art des Tests ab. Der Test sollte möglichst wenig unterschiedliche Einflüsse einwirken lassen, um besser Ursachen und Wirkungen verbinden zu können. Weiterhin ist es sinnvoll, möglichst wenig zufällige Ereignisse zu benutzen, um reproduzierbare Ergebnisse zu bestimmen.

nothingTest Die Testklasse *NothingTest* wurde als erster Test entwickelt, um eine Beispielimplementation anzugeben. Er führt in seiner Testroutine keine Aktionen aus und gibt direkt ein Standardergebnis zurück.

fileIO Dieser Test ist dazu gedacht, die Güte des *wear leveling* herauszufinden. Es wird eine einzelne Datei fester Größe[1] mit zufälligen Daten beschrieben und ausgelesen. Ein Dateisystem, das *wear leveling* benutzt, verteilt diese Zugriffe auf dem gesamten Speicher, um so einzelne Bereiche zu schonen. Das Ergebnis des Tests ist die Anzahl der erfolgten Zugriffe, bevor die Datei Fehler enthielt oder das Dateisystem beschädigt wurde.

fileIO_SEU Dieser Test bewertet die Fähigkeit des Dateisystemes, einzelne Bitfehler zu korrigieren. Dies wird erreicht, indem ein 5 KiB[2] großer Block mit Zufallsdaten gefüllt wird, und je Runde an fünf zufälligen Positionen im gesamten Flash Bitkipper mithilfe der Klasse *cause::induceSEU* injiziert werden (in dieser Implementation dementsprechend verteilt auf 528 KiB). Anschließend wird der Block ausgelesen. Enthält er Fehler, oder das Dateisystem kann die Datei nicht lesen, wird der Test beendet. Ansonsten werden die Daten wieder in die Datei geschrieben. In diesem Test akkumulieren sich die Fehler innerhalb der Datei nicht, da, falls im vorherigen Lauf Bitfehler wiederhergestellt werden konnten, sie korrigiert und zurückgeschrieben wurden. War eine Korrektur nicht möglich, wird der Test abgebrochen. Deshalb gibt es in der Datei maximal fünf Bitfehler gleichzeitig. Da dieser Test stark von zufälligen Ereignissen abhängt, wird der Test mit einem Skript mehrmals ausgeführt, und das Ergebnis an eine Datei angehängt. Durch das selbstgeschriebene Programm *ergebnissortierung.c* wird diese Datei eingelesen, die Ergebnisse sortiert, und formatiert ausgegeben. Das Ergebnis des Tests ist die Anzahl der insgesamt durchgeführten Bitkipper, bis der erste Fehler aufgetreten ist.

fileIO_SEL Dieser Test bewertet die Fähigkeit des Dateisystems, sowohl Bitfehler zu korrigieren, als auch *bad block*s zu erkennen. Der Test funktioniert wie der

[1] 2 KiB. Diese Größe kann an ein beliebiges Szenario angepasst werden.
[2] Ebenfalls zufällig gewählter, anpassbarer Wert.

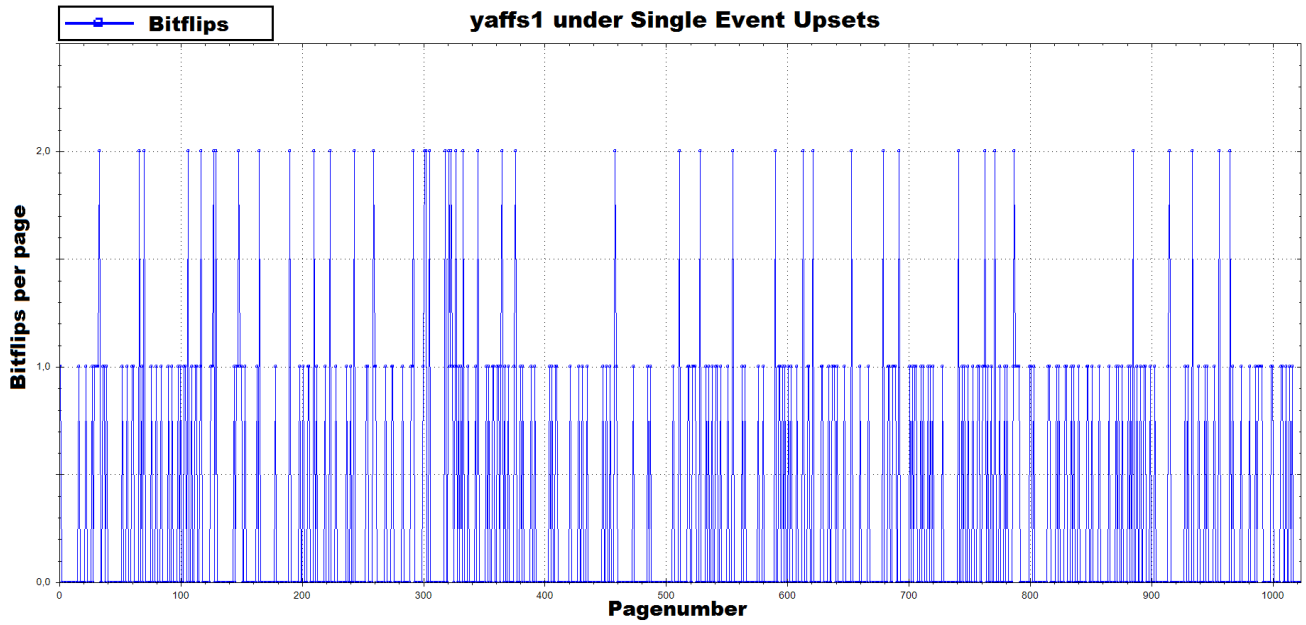


Abbildung 5.1: Gleichmäßige Verteilung der induzierten Bitkipper eines serialisierten Flashspeichers nach dem Test *fileIO_SEU*.

Test *fileIO_SEU*, mit dem Unterschied, dass mit der Klasse *cause::induceSEL* Single Event Latchups injiziert werden. Ein gutes Dateisystem würde geschriebene Daten verifizieren, und dadurch entdecken, dass ein *latchup* aufgetreten ist. Anschließend sollte es auf andere Positionen im Speicher ausweichen. Das Ergebnis ist die Anzahl der induzierten *latchup*s, bevor die Datei nicht mehr korrekt ausgelesen werden konnte.

fileIO_TID Dieser Test soll die Grenzen des Dateisystems bei steigender Fehlerrate erkennen. Obwohl kein entsprechendes Dateisystem implementiert ist, würde ein Dateisystem mit *cold redundancy* hier deutlich besser als Andere abschneiden, da ausgeschaltete Flashspeicher deutlich mehr Strahlung aushalten. In dem Test wird ein 15KiB großer Block mit zufälligen Daten gefüllt, die TID mit der Klasse *cause::increaseTID* in kleinen Schritten erhöht, und danach jeweils auf Bitfehler überprüft. Auch hier wird der gelesene Block wieder zurückgeschrieben, um akkumulierende Bitfehler zu vermeiden. Das Ergebnis des Tests ist die erreichte TID, bei der der erste Fehler stattgefunden hat.

maxFiles Dieser einfache Test ermittelt die Anzahl der im Wurzelverzeichnis erstellbaren Dateien. Dazu werden so lange Dateien geöffnet, mit wenigen Bytes beschrieben und wieder geschlossen, bis ein Fehler auftritt. Zurückgegeben wird die Anzahl der erfolgreich erstellten Dateien.

file_append Dieser Test ermittelt, wie groß eine Datei in einem Dateisystem werden kann. Es werden zyklisch 2KiB an eine Datei angehängt, bis das Dateisystem

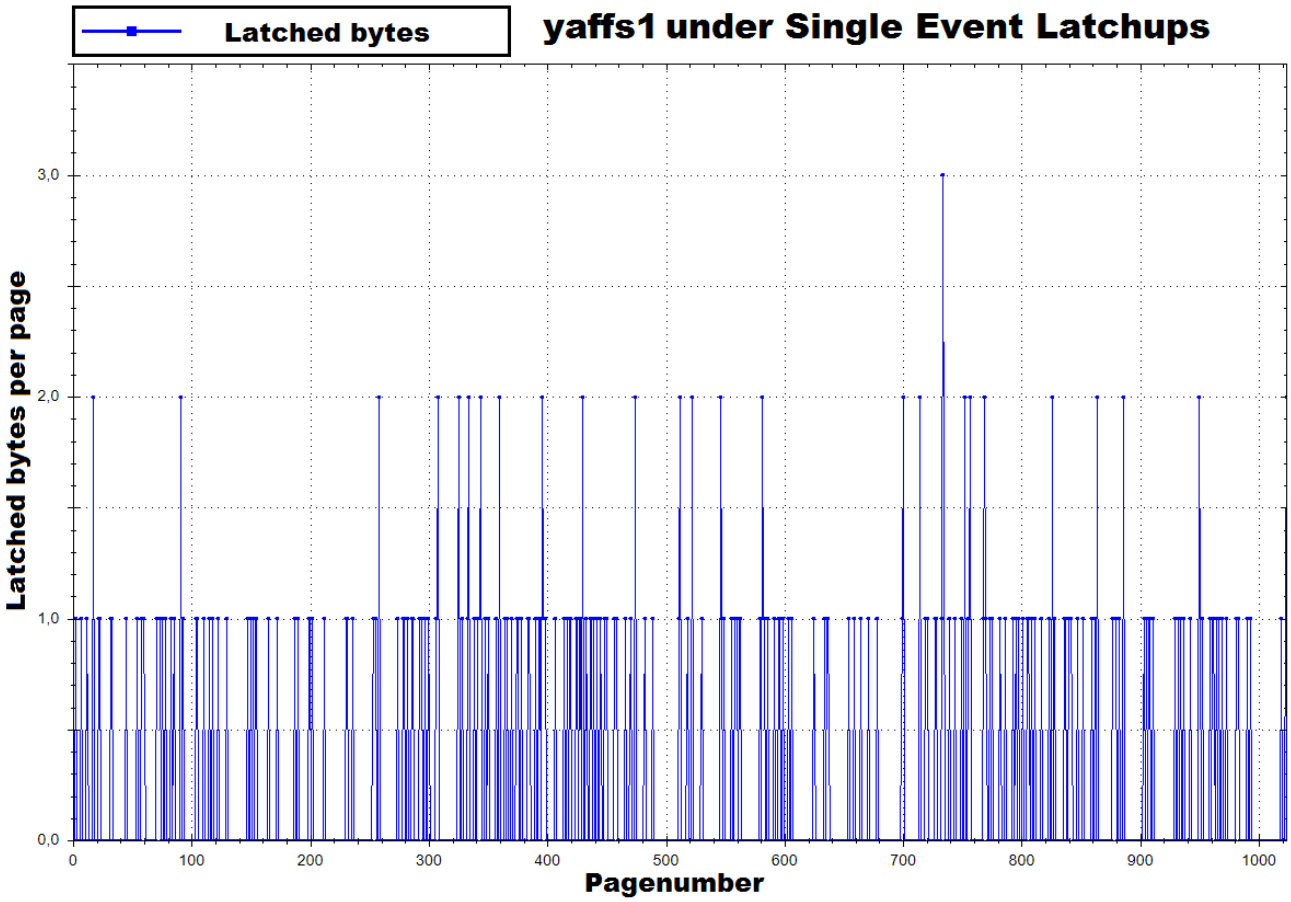


Abbildung 5.2: Gleichmäßige Verteilung der induzierten *latchups* eines serialisierten Flashspeichers nach dem Test *fileIO_SEL* mit *YAFFS1*.

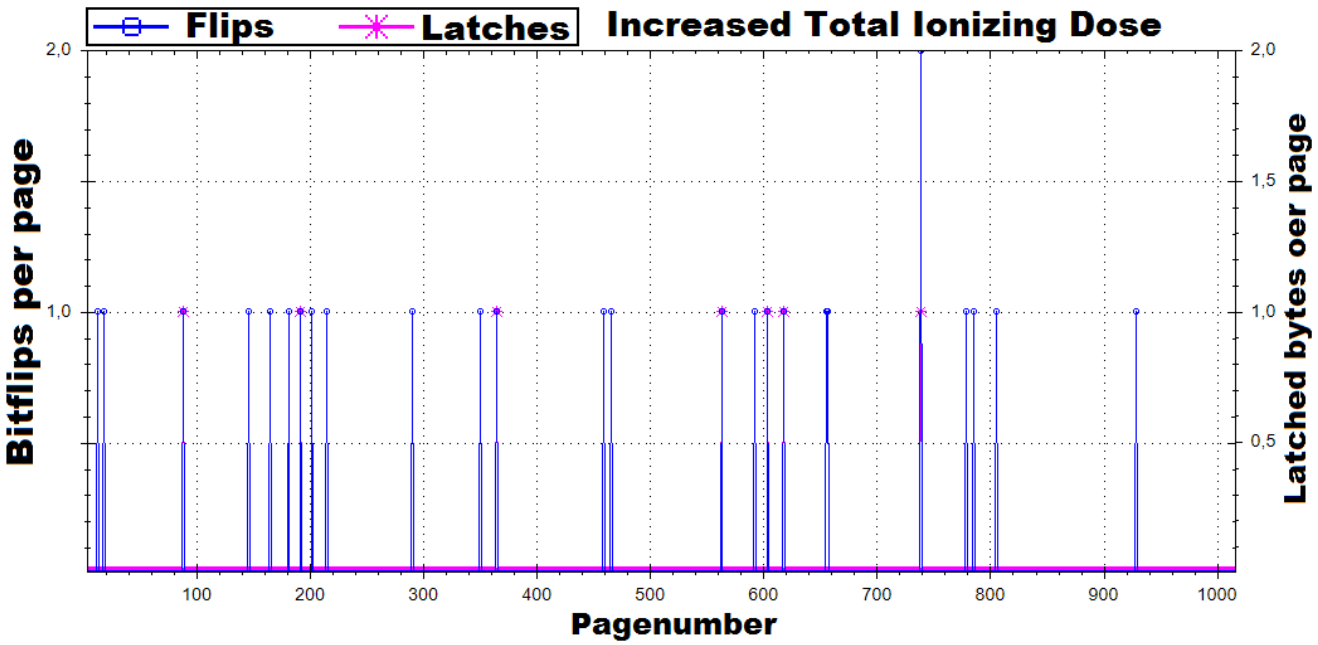


Abbildung 5.3: Ein serialisierter Flashspeicher nach dem Test *fileIO_TID* mit *YAFFS1*. Zu diesem Zeitpunkt ist die simulierte Strahlungsdosis 9560 Rad.

voll ist, oder andere Probleme auftauchen. Wenn das Dateisystem besonders schlecht *wear leveling* betreibt, kann es sein, dass vor dem Ende der Speicherkapazität bereits Blöcke beschädigt sind. Außerdem könnte der Speicher kleiner als die größte Datei sein. Das Ergebnis ist die Größe der Datei in Byte, bevor ein Fehler aufgetreten ist.

factoryDefects Dieser Test soll die Fähigkeit des Dateisystems, ab Werk defekte Blöcke zu erkennen, testen. Dazu werden mithilfe der Klasse *cause::factoryDefect* an vier zufälligen Positionen *bitline*s zerstört, was einen *latchup* in allen Pages eines Blockes an dieser Stelle bewirkt. Ein gutes Dateisystem würde bereits bei dem Formatieren die Blöcke nach *bad block markers* durchsuchen, oder jeden Block durch eine Schreib- und Leseoperation selbst überprüfen. Um festzustellen, ob ein Dateisystem diese Blöcke bemerkt hat, werden zwei Dateien erstellt, die insgesamt so groß sind, wie das Dateisystem den freien Speicherplatz angegeben hat[3]. Anschließend werden diese Dateien auf Bitfehler untersucht. Das Ergebnis des Tests ist die Anzahl der ab Werk defekten Blöcke, und ob das Dateisystem den Test bestanden hat. Dies ist der Fall, wenn sowohl die Dateien in der gewünschten Größe erstellt werden konnten, als auch keine Bitfehler in ihrem Inhalt aufgetreten sind.

[3]Um Ungenauigkeiten abzufangen, ist jede Datei ein Kilobyte (entspricht zwei Pages) kleiner, als sie laut Freispeicheranzeige maximal sein könnte.

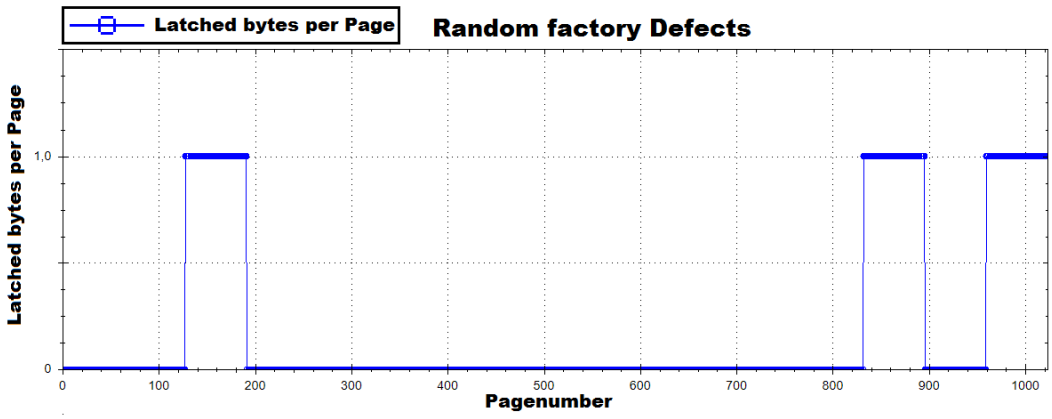


Abbildung 5.4: Ein Flashspeicher mit drei defekten Blöcken nach dem Test *factory-Defect* mit *YAFFS1*. Es werden die *latchups* dargestellt.

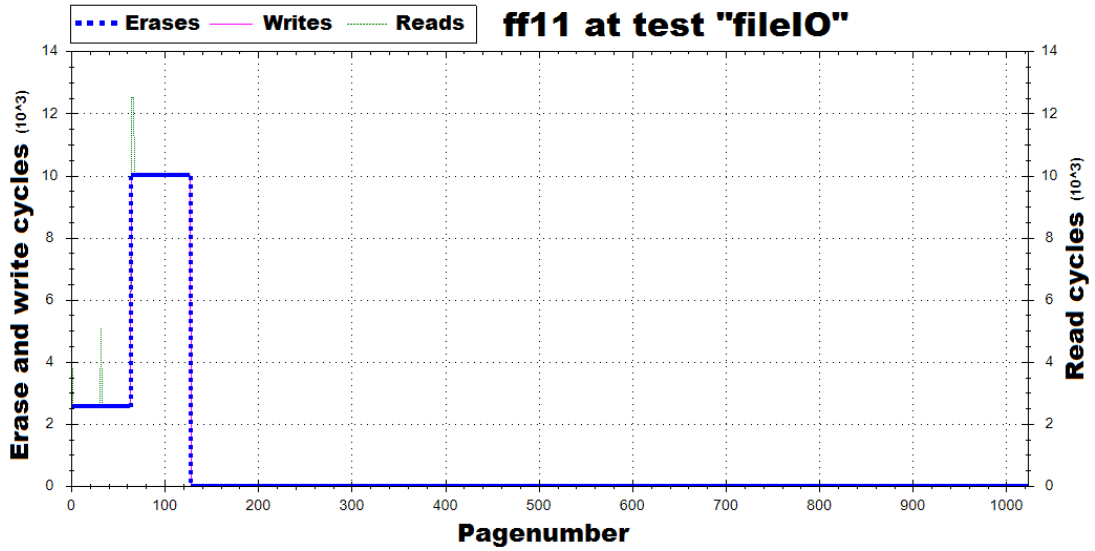
5.2 Testergebnisse

Auf der Basis implementierter Tests werden die Dateisysteme *FAT* und *YAFFS1* bewertet und das Verhalten beschrieben.

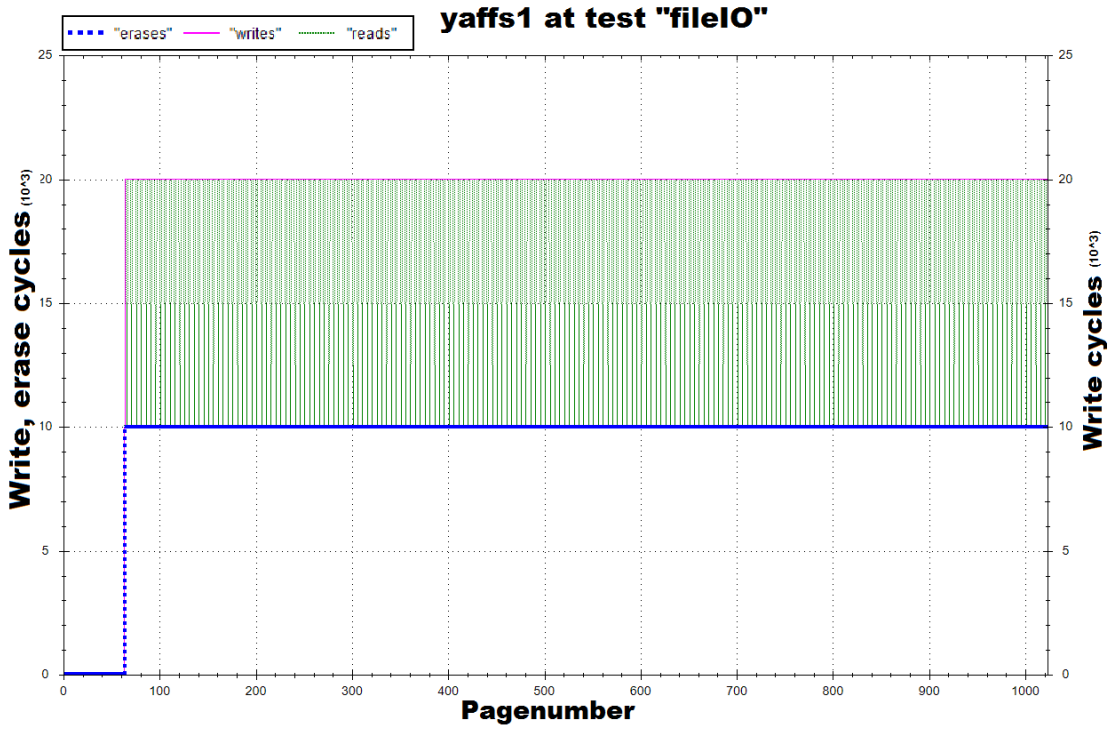
5.2.1 Wear Leveling

In Abb. 5.5 ist der Unterschied im *wear leveling* zwischen den beiden Dateisystemen gut zu erkennen. In diesem Test wurden die Ausfalleigenschaften verändert, um schnellere und eindeutigere Testdurchläufe zu gewährleisten. Die Lebensdauer aller Zellen wurde auf *10.000* Löschvorgänge beschränkt, daher ist die *erases* Kurve eine gerade Linie. In der Realität würden die Zellen im Schnitt um den Faktor 10 länger halten, und eine deutlich höhere Varianz aufweisen.
Bei Abb. 5.5(a), *FAT*, wird nur der erste Block, der die MFT enthält, und der zweite Block, der die Datei enthält, benutzt. Da sich die Datei nicht in ihrer Größe ändert, wird die MFT seltener benutzt. Der Dateiinhalt wurde ungültig, nachdem der Block abgenutzt war. Insgesamt konnten **2500** Schreibzugriffe stattfinden, was sich direkt auf die Treiberimplementation abbilden lässt, die pro Zugriff einen ganzen Block löscht. Um die erforderlichen vier Sektoren (Pages) zu überschreiben, wird der Block daher vier mal gelöscht. Dieses Verhalten entspricht in schlechten Fällen einem *flash unaware* FTL.
YAFFS1 (Abb. 5.5(b)) hingegen verteilt die Schreibzugriffe (vgl. Abschnitt 3.4) und erkennt durch Verifikation geschriebener Daten (Einlesen der Daten nach einem Zugriff) abgenutzte Blöcke, und verwendet sie danach nicht mehr. Durch diese Strategie kann eine einzelne Datei auf dem gesamten Flashspeicher[4] verteilt werden. Die Beendigung des Tests erfolgt durch die Meldung, dass kein freier (unbeschädigter) Speicherplatz mehr alloziert werden konnte. Die Datei konnte **1920190** mal beschrieben

[4]Bis auf einen Block, der für *garbage collection* reserviert ist. Empfohlen sind 3, allerdings ist dies bei einem kleinen Speicher nicht sinnvoll.



(a) FAT

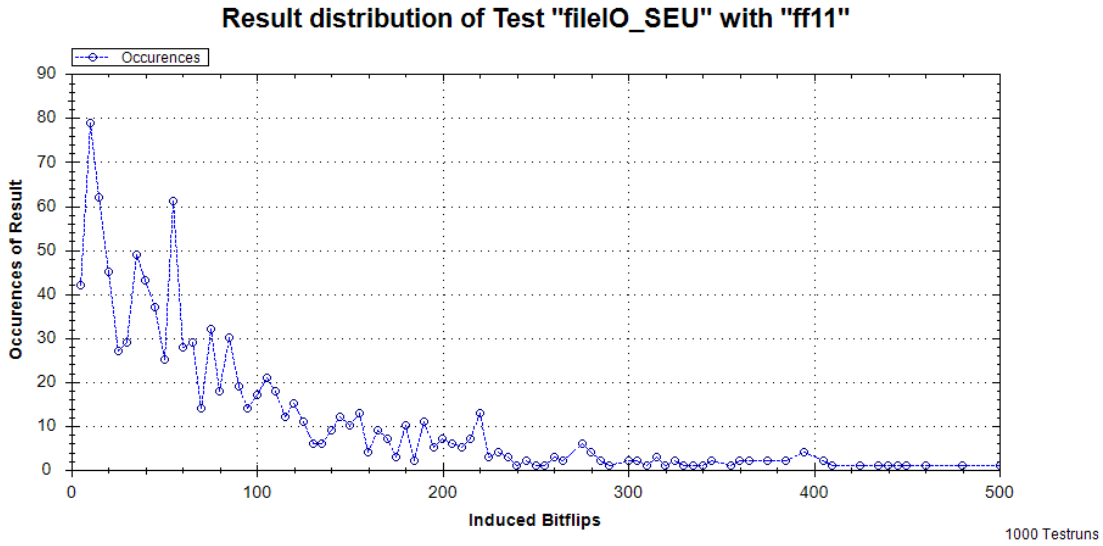


(b) YAFFS1

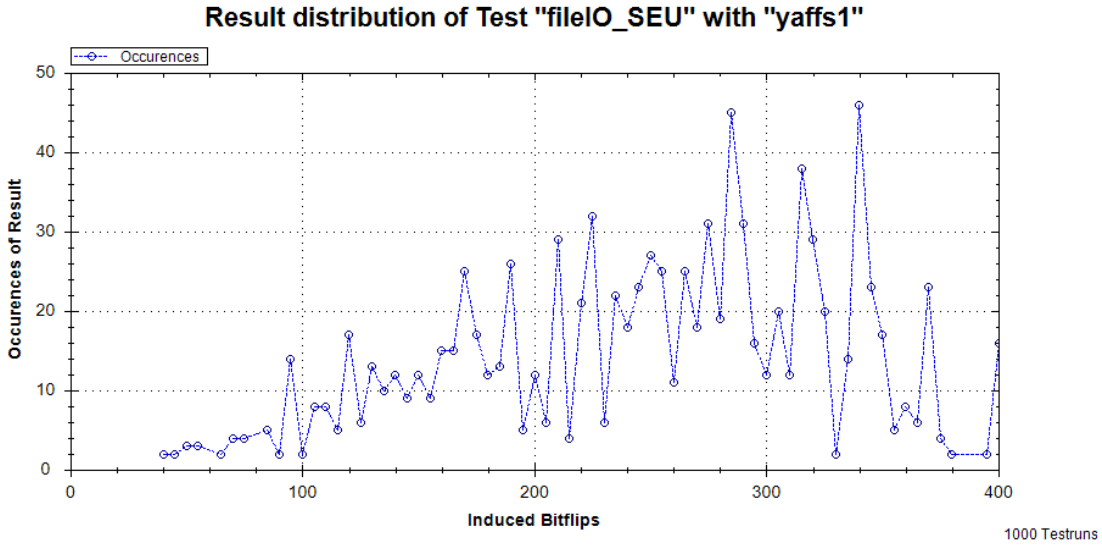
Abbildung 5.5: Diagramm der serialisierten Flashspeicher jeweils nach dem Test *fileIO*. Auf der X-Achse verteilen sich die Pages, auf der Y-Achse links werden Lösch- und Schreibzyklen aufgelistet, auf der Rechten Y-Achse ist die Anzahl der Lesezugriffe dargestellt.

werden, was eine Verbesserung von rund 76800% gegenüber *FAT* entspricht. Bei einer kleineren Datei oder einem größeren Speicher wäre diese Verbesserung noch größer, da die Datei auf mehr Platz verteilt werden kann.

5.2.2 Bitfehlertoleranz



(a) FAT



(b) YAFFS1

Abbildung 5.6: Die Verteilung der Ergebnisse des Tests *fileIO_SEU* mit *YAFFS1* und *ff11*. Eine gewisse Stückelung ist vorhanden, da in jedem Testzyklus 5 Fehler injiziert wurden.

Auf Abb. 5.6 ist der Unterschied der Fehleranfälligkeit zwischen den beiden Dateisystemen gut zu erkennen. *ff11* erreicht im Mittelwert nach 1000 Testläufen **90,7** Bitfehler, bis die Datei Fehler enthält, obwohl *FAT* keine Fehlerkorrektur vorsieht. Das liegt an der Größe der getesteten Datei, da nur Bitfehler, die in der (statischen) Region der Datei oder der MFT vorkommen, eine Auswirkung haben.

Wäre die Datei größer, würde sie früher Defekte aufweisen. Das ist auch der Grund, warum in dieser Verteilung teilweise sehr hohe Werte (Maximum **1040** Bitflips) vorkommen. In diesem Fall ist der überwiegende Teil der SEUs in unbenutzten Regionen vorgekommen. Der Median liegt liegt weit von dem Mittelwert entfernt bei **60** Bitfehlern, und in ≈ 4% aller Fälle ist die Datei bereits im ersten Durchlauf nach 5 Bitfehlern verändert worden.
YAFFS1 hingegen erreicht im Schnitt **247,8** Bitfehler, was einer Verbesserung von ≈ 173% entspricht. Das ist auf den eingebauten ECC zurückzuführen, der einen Bitfehler pro Page korrigieren kann[5]. Somit müssen zwei Bitflips in einer Page stattfinden, um Daten darin ungültig zu machen. Da *YAFFS* wegen des *wear leveling* die Daten auf den kompletten Speicher verteilt, kann ein Bitkipper an jeder Position potentiell die Datei oder die Dateisystemstruktur treffen. Das bewirkt eine deutlich geringere Streuung der Testergebnisse (Median: **255** Bitflips, nur 2,7 Flips unter dem Mittelwert), da die Häufung der Fehler an einer Position keinen Unterschied macht. In 1000 Testresultaten ist kein Ergebnis unter **40** Bitflips vorgekommen, der höchste Wert **400** wurde 16 mal erreicht.

5.2.3 Bad Block Erkennung während des Betriebes

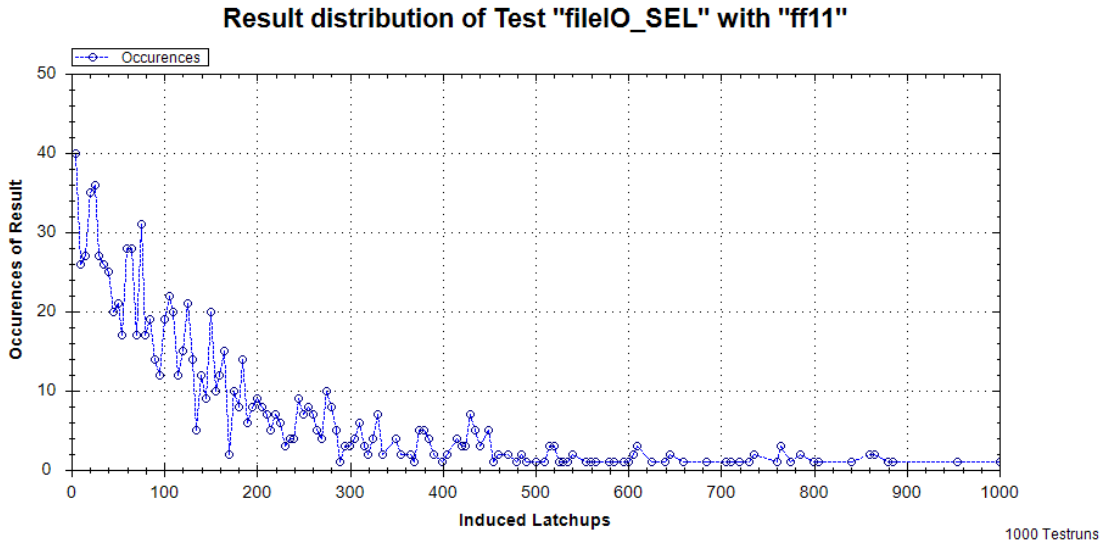
Das Abschneiden beider Dateisysteme ist in diesem Test vergleichbar, aber etwas besser als im Test *fileIO_SEU*. Das hängt damit zusammen, dass ein *latchup* in einem Bit nur dann auffällt, wenn es ungleich 0 war. Da *FAT* bei der gegebenen Dateigröße nur einen kleinen Teil des gesamten Speichers benutzt, ist auch hier die Abweichung des Medians (**105** *latchups*) gegenüber dem Mittelwert (**169,6** *latchups*) höher als bei *YAFFS1* (Median: **260**, Mittelwert: **251**). Da selbst die benutzten Teile von *FAT* viele Nullen beinhalten, fällt der *latchup* seltener auf, womit sich *ff11* gegenüber *YAFFS1* deutlich verbessert, aber immer noch ≈ 48% schlechter ist. *YAFFS* verbessert sich nur geringfügig, da zwar *latchup*s erkannt und umgangen werden, aber bei der großen Menge an Fehlern keine funktionsfähigen Blöcke übrig bleiben.

5.2.4 Verhalten bei erhöhter TID

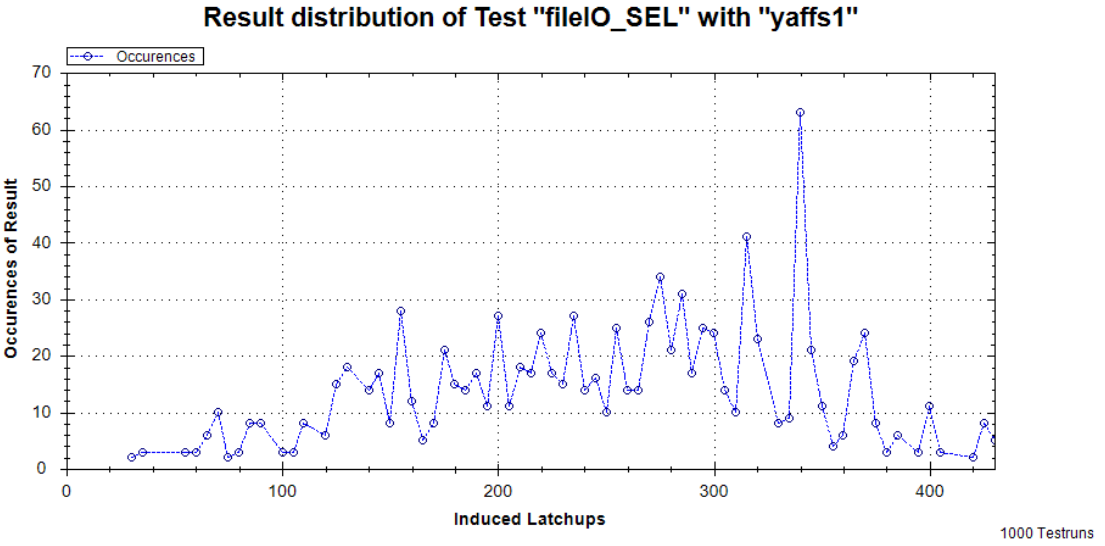
In diesem Test wurden die Eigenschaften des Flashspeichers so definiert, dass im Durchschnitt ein Bit 30 kRad aushält, mit einer Varianz von 5kRad. Da es knapp 530.000 Bit in der Simulation gibt, und schon wenige Bitfehler bzw. *latchup*s (in den Testläufen im Schnitt zwischen 60 und 255) den Abbruch auslösen, wird dieser Wert in der Praxis nie erreicht.
ff11 erreicht im Durchschnitt eine TID von **8925** Rad. Dies entspricht je nach Bauart der Realität[6]. Die Verteilung ist bei beiden Dateisystemen sehr in Richtung geringer

[5] *YAFFS* im Allgemeinen kann drei Bitfehler pro Page erkennen und Einen korrigieren.

[6] Die ersten Bitfehler treten bei NAND-Flash meistens zwischen 8-14 kRad auf (Nguyen u. a. (1998), Nguyen u. Scheick (2003)). Die Parametrisierung kann aber an den verwendeten Flashspeicher angepasst werden. Z.B. entspricht die Konfiguration *Mittlere TID: 48 kRad, Varianz: 5500*

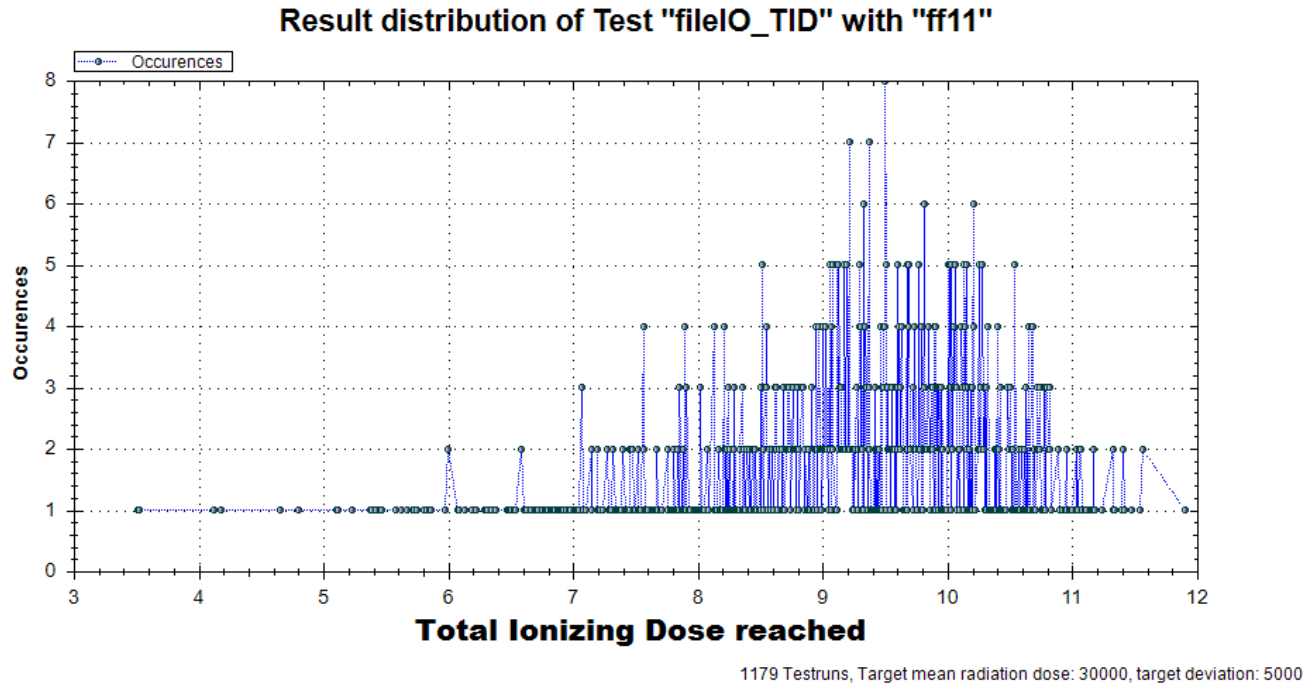


(a) FAT

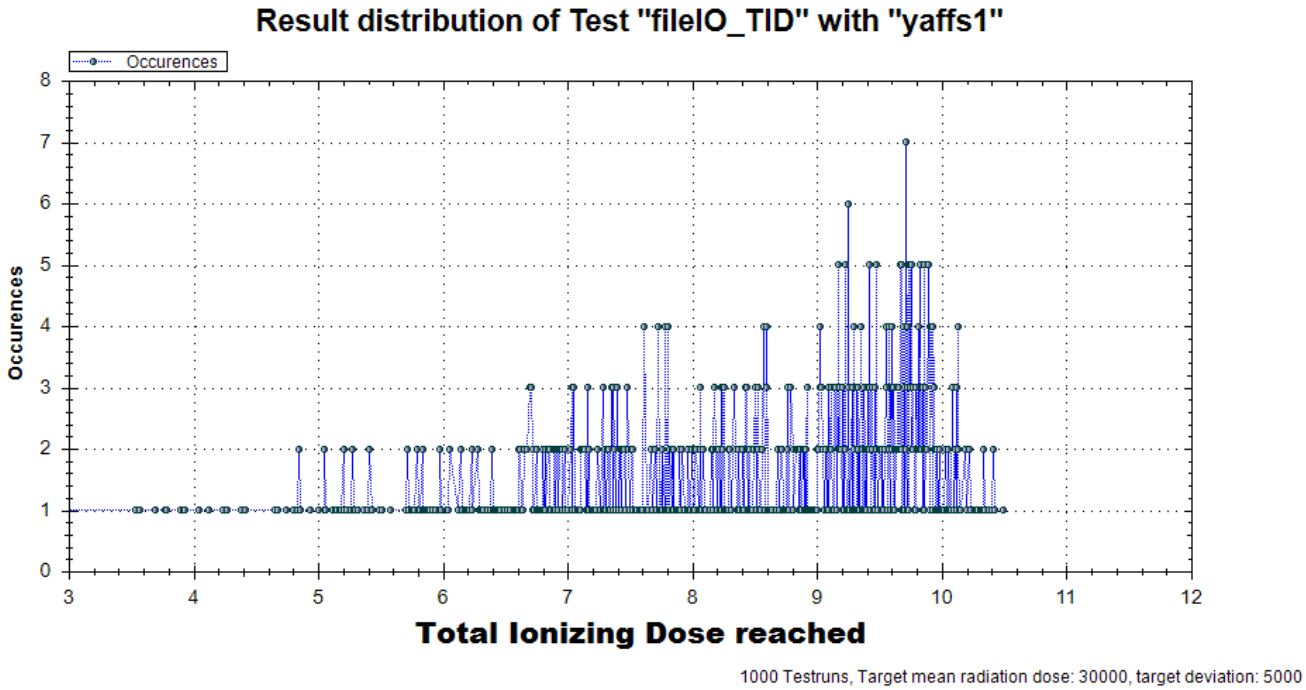


(b) YAFFS1

Abbildung 5.7: Die Verteilung der Ergebnisse des Tests *fileIO_SEL* mit *YAFFS1* und *ff11*. Eine gewisse Stückelung ist vorhanden, da in jedem Testzyklus 5 Fehler injiziert wurden.



(a) FAT



(b) YAFFS1

Abbildung 5.8: Die Verteilung der Ergebnisse des Tests *fileIO_TID* mit *YAFFS1* und *ff11*. Der Test wurde jeweils 1000 mal ausgeführt, und die Ergebnisse anschließend sortiert.

TID gestreckt, da bei *ff11* ein, bei *YAFFS1* zwei, *latchup*s oder Bitflips in der Speicherposition der Datei ausreichen, um den Test abbrechen zu lassen. *YAFFS1* bricht im Schnitt nach **8318** Rad ab, was ≈ 7% schlechter als *ff11* ist. Dies hat den Grund, dass *YAFFS* die Datei über den gesamten Speicher verteilt, und *YAFFS* auch merkt, wann ein Block beschädigt ist, und ihn nicht weiter verwendet. In ≈ 81% der Fälle wurde der Test nicht aufgrund von Bitfehlern in der Datei beendet, sondern weil *YAFFS* keine weiteren Blöcke zum Schreiben allozieren konnte. Wird der Test verändert, sodass erst nach einem Dateifehler abgebrochen wird, kommt *YAFFS1* im Schnitt auf **8729,8** Rad.

5.2.5 Erstellbare Dateien

Auf Abb. 5.9 ist die unterschiedliche Strategie zwischen einer zentralen MFT (*FAT*, (a)) und der dezentralen Verteilung (*YAFFS*, (b)) zu erkennen. Bei *FAT* wird überwiegend der erste Block beansprucht, da dort neue Dateien und deren Positionen eingetragen werden. *ff11* kann **512** Dateien in das Wurzelverzeichnis legen, was sich mit der Spezifikation von *FAT16* mit kleiner MFT deckt. *YAFFS1* hingegen kommt nur auf **447** Dateien in diesem Speicher, was allerdings nicht von einer vorher festgelegten Größe einer MFT liegt, sondern dass die Struktur zur Verwaltung der Dateien mit der Anzahl der Dateien selbst wächst. Wäre der Speicher größer, könnten mehr Dateien gespeichert werden. Da *YAFFS1* nicht komprimiert, könnten die Dateien wie bei *ff11* auch, größer sein, ohne dass sich am Limit etwas ändert. Aus der Benutzung des ersten Blockes in Abb. 5.9(b) kann geschlossen werden, dass der *garbage collector* bereits gelaufen ist.

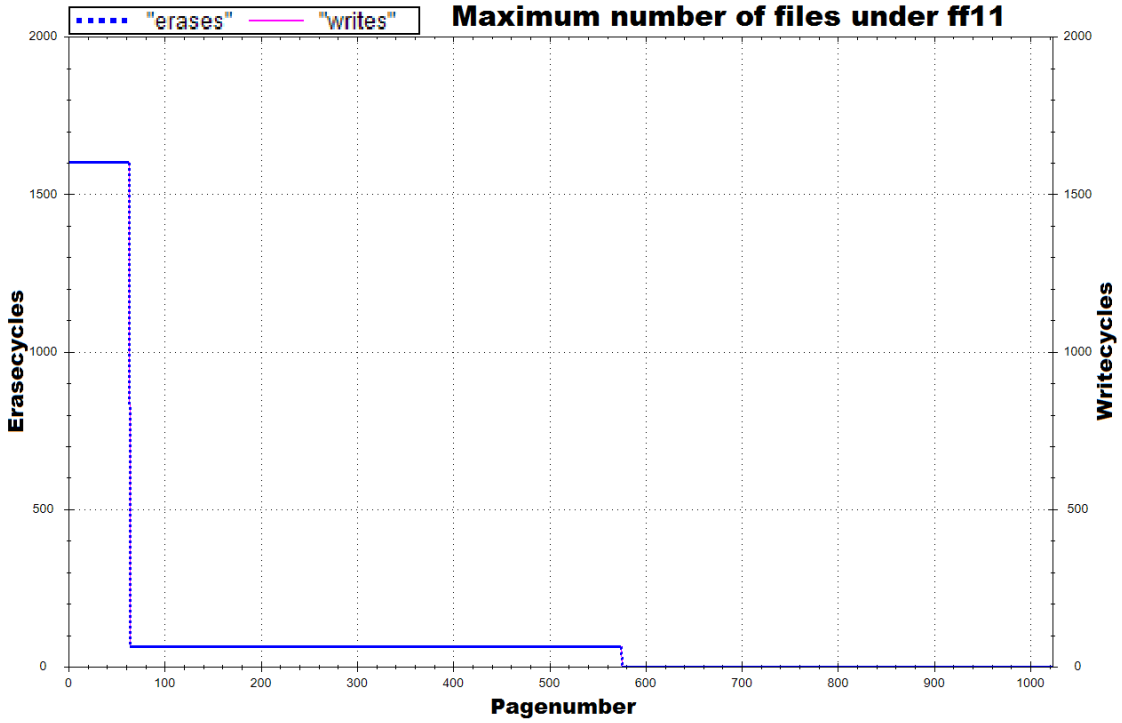
5.2.6 Maximale Dateigröße

Dieser Test ähnelt von der Abnutzung sehr Abschnitt 5.2.5. *ff11* schneidet hier mit **491520** geschriebenen Byte ≈ 7% besser ab als *YAFFS1* mit **458767** Byte (von insgesamt verwendbaren 512 KiB + 32 KiB OOBA), da *FAT* selbst ohne Verwendung der OOBA durch die zentrale Dateiverwaltung viel Platz spart.

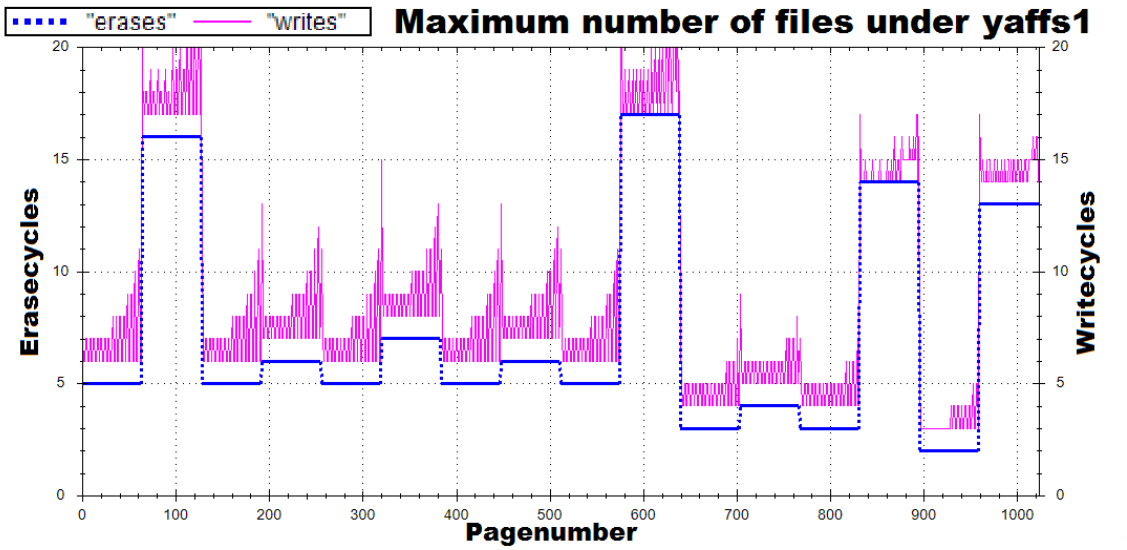
5.2.7 Bad Block Erkennung bei Inbetriebnahme

Da der Standard *FAT* keine *bad blocks* kennt, schafft es *ff11* nicht, die fehlerhaften Blöcke zu umgehen und meldet mehr Speicherplatz als tatsächlich funktionstüchtig ist. *YAFFS1* hingegen scannt bei dem *mounting* alle Blöcke nach dem *bad block marker*, und meldet daher bei drei defekten *bad blocks* **327680** statt 425472 freie Bytes. Moderne MLC NAND-Speicher dürfen i.d.R. bis zu 5 defekte Blöcke im Auslieferungszustand besitzen (Mutlu, 2014).

Rad ungefähr dem in Nguyen u. Scheick (2003) geschilderten ,,Intel 256 MiB MLC" Flashspeichern.

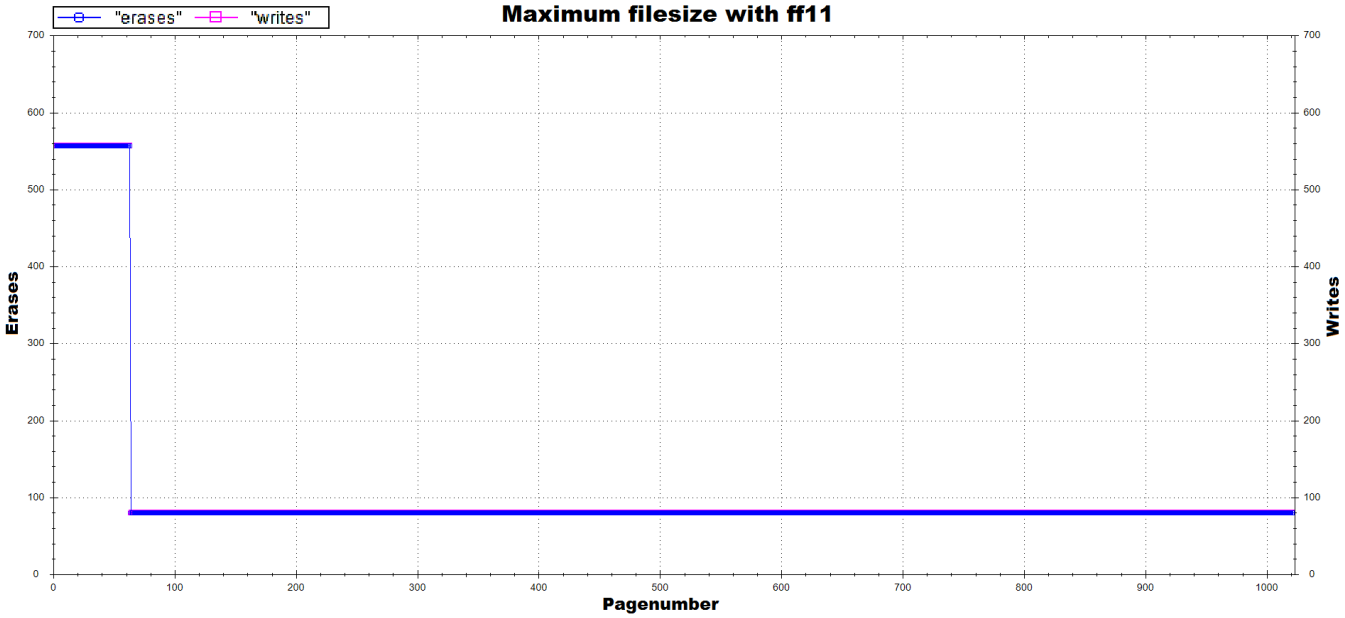


(a) FAT

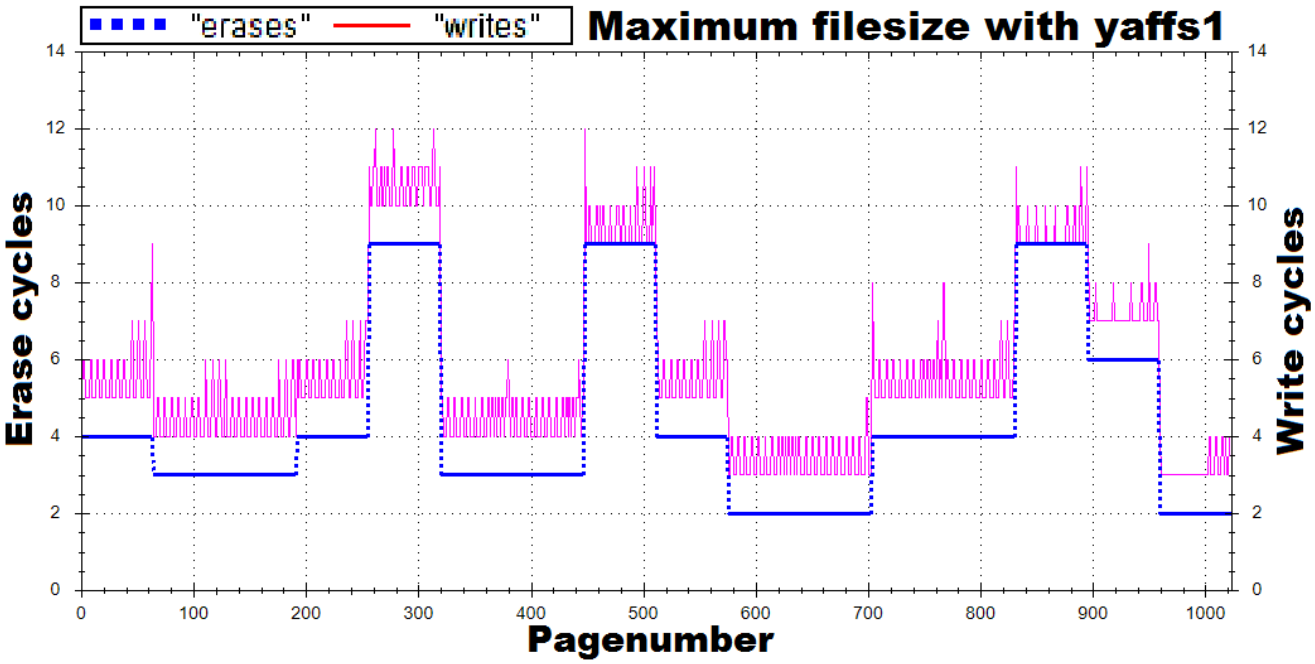


(b) YAFFS1

Abbildung 5.9: Löschzyklen und Schreibzugriffe der serialisierten Flashspeicher jeweils nach dem Test *file_append*.



(a) FAT



(b) YAFFS1

Abbildung 5.10: Löschzyklen und Schreibzugriffe der serialisierten Flashspeicher jeweils nach dem Test *file_append*.

Kapitel 6

Fazit

6.1 Zusammenfassung

Im Laufe der Bachelorarbeit wurde eine Simulationsumgebung entwickelt, mit denen das Verhalten beliebiger Dateisysteme unter den häufigsten im Weltraum herrschenden Fehlerquellen quantitativ bewertet werden kann. Um dies zu erreichen, wurden eine Reihe von Tests entwickelt, Fehlerursachen klassifiziert und eine NAND-Flash Simulation implementiert. Die Simulation ist vollständig parametrierbar, um der Realität je nach Speicherchip so genau wie möglich zu entsprechen. Es wurden für zwei existierende Dateisysteme Treiber entwickelt, und diese Dateisysteme bewertet. Weiterhin wurden existierende Dateisysteme verglichen, und die Grundlage für das Entwickeln eines robusten Dateisystems geschaffen, das es ermöglichen kann, günstigeren Hauptspeicher in Anwendungsgebiete mit hoher Strahlung, wie z.B. der Raumfahrt, einzusetzen.

6.2 Ausblick

Da die Zeit der Bachelorarbeit begrenzt war, sind einige Optimierungsmöglichkeiten offengeblieben:

Tests und Fehlerklassen trennen Um die Simulationsumgebung dynamischer zu gestalten, können die Tests von Fehlerklassen getrennt werden. Dann würde ein Test wie z.B. *fileIO* mit dem Einspielen von Bitflips oder unter hoher TID kombiniert werden können, ohne dass er neu geschrieben werden müsste.

Größere Testbibliothek Um noch mehr Aspekte eines Dateisystems zu testen, können weitere Tests hinzugefügt werden. So könnte ein Test z.B. weitere Limits eines Dateisystems erforschen, um auch die Integrität der Implementierung zu überprüfen.

Tests in Skriptsprache Wenn die Tests dynamisch interpretiert werden würden, also die Simulationsumgebung für z.B. Perl Schnittstellen bietet, können

Tests schneller programmiert und angepasst werden. Dadurch müsste das komplette Programm nicht immer neu kompiliert und gelinkt werden.

Externes Interface für Dateisysteme Um zu ermöglichen, dass Dateisysteme anderer Programmiersprachen getestet werden können, und bei nicht jeder Änderung alle anderen Dateisysteme und die Simulationsumgebung neu kompiliert und gelinkt werden müssen, könnte die Simulationsumgebung eine externe Schnittstelle anbieten, mit der ein Dateisystem z.B. als *dynamically linked library* durch ein Pluginsystem zur Laufzeit eingebunden werden kann.

Visualisierung zur Laufzeit Um die Mechanismen eines Dateisystems zu studieren, oder um Fehler in ihnen zu finden, könnte es helfen, sich die *FlashCell* visualisieren zu lassen. Durch *debugging* Programme kann die Ausführung des Dateisystems pausiert werden, und der Flashspeicher ausgelesen oder verändert werden.

Anhang A

Verwendete Programme

Zur Visualisierung der Statistiken wurde der kostenlose CSV-Plotter *DatPlot*[1] benutzt.
Die UML-Diagramme wurden mit dem kostenlosen Werkzeug *UMLet*[2] erstellt.
Um die Simulationsumgebung möglichst portabel zu halten, ist das *CMake*[3] Buildsystem benutzt worden.

[1] http://www.datplot.com
[2] http://www.umlet.com/
[3] https://cmake.org/

Anhang B

Inhalt der beiliegenden CD

Die beigelegte CD beinhaltet sowohl alle im Rahmen der Bachelorarbeit entstandenen Programme als auch die Testergebnisse und dieses Dokument in digitaler Form.

```
/
├── Bachelorarbeit.pdf
├── SATFON/
│   ├── ds/
│   ├── errInduce/
│   ├── interface/
│   ├── misc/
│   ├── simu/
│   ├── tests/
│   └── main.cpp
└── Testergebnisse/
    ├── serialized/
    │   └── ...
    └── statistics/
        └── ...
```

Der Quellcode der Simulationsumgebung befindet sich in dem Ordner **SATFON**. Es enthält alle Quelldateien inkl. der beiden Statistikanalysetools und entsprechender *CMakefiles*. Um ihn zu kompilieren, muss die Buildkonfiguration mithilfe von *CMake* in einem beliebigen Verzeichnis erstellt werden. Anschließend kann es gebaut werden. Die Dateisysteme mit ihren Treibern liegen in **ds**, während ihre Abstraktionsinterfaces in **interface** liegen. Die implementierten Fehlerklassen sind in **errInduce**. **misc** beinhaltet Hilfsprogramme wie die *ergebnissortierung*, *instanceRegistry* und Windows-Skripte zum mehrmaligen Ausführen spezieller Tests. In **tests** liegen alle implementierten Tests.
Der Ordner **Testergebnisse** beinhaltet alle Ergebnisse, die für die Auswertungen in Abschnitt 5.2 benötigt wurden. Sie sind aufgeteilt in die Ordner **serialized**, in dem alle einzelnen Testergebnisse sind, und **statistics**, in dem alle statistisch ausgewertete Resultate sind.

Abbildungsverzeichnis

Tabellenverzeichnis

Akronyme

AMD Algebraic Manipulation Detection. 25

COTS Commercial Off-The-Shelf. 4

CPU Central Processing Unit. 16

ECC Error Correcting Code. 10, 14, 17, 20–25, 28, 52

EDC Error Detecting Code. 10, 21

EEPROM Electrically Erasable and Programmable Read-Only Memory. 8

EPROM electrically programmable read-only memory. 8

FTL Flash Translation Layer. 6, 7, 14, 15, 17, 24–27, 50

HDD Hard Disk Drive. 4, 8, 15, 17

MFT Master File Table. 27, 50, 51, 57

MLC Multi Level Cell. 9, 55, 59

MOSFET Metal Oxide Semiconductor Field-Effect Transistor. 9–13, 63

MTD Memory Technology Device. 23

MTTF Mean Time To Failure. 16, 25

OOBA Out Of Band Area. 10, 14, 15, 21, 23, 44, 58

POSIX Portable Operating System Interface X. 18, 19, 65

radhard radiation hardened components. 4

RAID Redundant Array of Independent Disks. 15–17, 21, 24–27, 35

RTOS RealTime Operating System. 21

SATFON Suite for Automated Testing of Filesystems On Nandflash. 5, 27, 37, 39

SEE Single Event Effect. 13

SEL Single Event Latchup. 13, 14, 25, 31, 43, 45

SEU Single Event Upset. 13, 14, 41, 43, 52

SLC Single Level Cell. 9, 44

SSD Solid State Drive. 7

TID Total Ionizing Dose. 13, 14, 20, 42–44, 46, 55

Literaturverzeichnis

[Bez u. a. 1988] Bez, R. ; Camerlenghi, E. ; Modelli, A. ; Visconti, A.: Introduction to flash memory. In: *IEEE Transactions on Power Systems blank page* 76 (1988), Nr. 4, 489-502. http://dx.doi.org/10.1109/JPROC.2003.811702. – ISSN 00189219

[Bucy u. a. 2008] Bucy, John S. ; Schindler, Jiri ; Schlosser, Steven W. ; Ganger, Gregory R.: *The DiskSim Simulation Environment Version 4.0 Reference Manual*. 2008

[Chen u. a. 2006] Chen, Tianzhou ; Wang, Xiangsheng ; Hu, Wei ; Duan, Wei: A New Type of NAND Flash-Based File System: Design and Implementation. In: *Author Index* (2006). http://dx.doi.org/10.1109/WiCOM.2006.408. ISBN 1424405173

[Cooperation of open taskgroups 2009] Cooperation of open taskgroups: *Portable Operating System Interface (POSIX)*. http://dx.doi.org/10.1109/IEEESTD.2009.5393893. Version: 2009

[Corbett u. a. 2004] Corbett, Peter ; English, Bob ; Goel, Atul ; Grcanac, Tomislav ; Kleiman, Steven ; Leong, James ; Sankar, Sunitha: Row-diagonal parity for double disk failure correction. In: *In Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, 2004, S. 1–14

[Gao u. a. 2010] Gao, Yan ; Meister, Dirk ; Brinkmann, André: Reliability Analysis of Declustered-Parity RAID 6 with Disk Scrubbing and Considering Irrecoverable Read Errors. In: *A Summary of Granular Computing System Vulnerabilities: Exploring the Dark Side of Social Networking Communities* (2010), 126-134. http://dx.doi.org/10.1109/NAS.2010.11. ISBN 9781424481330

[Im u. Shin 2011] Im, Soojun ; Shin, Dongkun: Flash-Aware RAID Techniques for Dependable and High-Performance Flash Memory SSD. In: *Call for Papers* 54 (2011), Nr. 1, 80-92. http://dx.doi.org/10.1109/TC.2010.197. – ISSN 00189340

[Jain u. Lee 2006] Jain, S. ; Lee, Yann-Hang: Real-time support of flash memory file system for embedded applications. In: *Author Index* (2006), 6 pp. http://dx.doi.org/10.1109/SEUS-WCCIA.2006.35. ISBN 0769525601

[Jedrak 2011] Jedrak, Michal ; S.A, Evatronix (Hrsg.): *NAND Flash memory in embedded systems*. http://www.design-reuse.com/articles/24503/nand-flash-memory-embedded-systems.html. Version: September 2011, Abruf: 5. Juni 2016. – [Online]

[Jung u. a. 2012] Jung, Myoungsoo ; Wilson, Ellis H. ; Donofrio, David ; Shalf, John ; Kandemir, Mahmut T.: NANDFlashSim: Intrinsic latency variation aware NAND flash memory system modeling and simulation at microarchitecture level. In: *IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012, April 16-20, 2012, Asilomar Conference Grounds, Pacific Grove, CA, USA*, 2012, 1–12

[Kang u. Miller 2009] Kang, Yangwook ; Miller, Ethan L.: *Adding Aggressive Error Correction to a High-Performance Compressing Flash File System*. 2009

[Kim u. a. 2013] Kim, Jaeho ; Lee, Jongmin ; Choi, Jongmoo ; Lee, Donghee: Improving SSD reliability with RAID via Elastic Striping and Anywhere Parity. In: *Dynamic Reconfiguration of Wireless Sensor Networks to Support Heterogeneous Applications* (2013), 1-12. http://dx.doi.org/10.1109/DSN.2013.6575359. – ISSN 97814676364720153008889

[Lee u. a. 2011] Lee, Sehwan ; Lee, Bitna ; Koh, Kern ; Bahn, Hyokyung: A Lifespan-aware Reliability Scheme for RAID-based Flash Storage. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. New York, NY, USA : ACM, 2011 (SAC '11). – ISBN 978–1–4503–0113–8, 374–379

[Lee u. a. 2009] Lee, Yangsup ; Jung, Sanghyuk ; Song, Yong H.: FRA: A Flash-aware Redundancy Array of Flash Storage Devices. In: *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. New York, NY, USA : ACM, 2009 (CODES+ISSS '09). – ISBN 978–1–60558–628–1, 163–172

[Lim u. Park 2006] Lim, Seung-Ho ; Park, Kyu-Ho: An efficient NAND flash file system for flash memory storage. In: *Computers, IEEE Transactions on* 55 (2006), July, Nr. 7, S. 906–912. http://dx.doi.org/10.1109/TC.2006.96. – DOI 10.1109/TC.2006.96. – ISSN 0018–9340

[Lin u. a. 2006] Lin, Chuan-Sheng ; Chen, Kuang-Yuan ; Wang, Yu-Hsian ; Dung, Lan-Rong: A NAND Flash Memory Controller for SD/MMC Flash Memory Card. In: *Author index* (2006). http://dx.doi.org/10.1109/ICECS.2006.379716. ISBN 1424403944

[Luo u. a. 2013] Luo, Pei ; Wang, Zhen ; Karpovsky, Mark: Secure NAND Flash Architecture Resilient to Strong Fault-Injection Attacks Using Algebraic Manipulation Detection Code. (2013). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.310.2968

[Manning 2010] Manning, Charles: How YAFFS works. (2010). http://dubeyko.com/development/FileSystems/YAFFS/HowYaffsWorks.pdf

[Micron Technology 2008] Micron Technology, Inc.: *Wear-Leveling Techniques in NAND Flash Devices*. https://www.micron.com/~/media/documents/products/technical-note/nand-flash/tn2961_wear_leveling_in_nand.pdf. Version: Oktober 2008, Abruf: 5. Juni 2016. – [Online]

[Mutlu 2014] Mutlu, Onur: *Error Analysis and Management for MLC NAND Flash Memory*. 2014

[Nguyen u. a. 1998] Nguyen, D.N. ; Lee, C.I. ; Johnston, A.H.: Total ionizing dose effects on flash memories. In: *Introduction to microcontrollers - Part 2* (1998), 100-103. http://dx.doi.org/10.1109/REDW.1998.731486. ISBN 0780351096

[Nguyen u. Scheick 2003] Nguyen, D.N. ; Scheick, L.Z.: TID, SEE and radiation induced failures in advanced flash memories. In: *Author index* (2003), 18-23. http://dx.doi.org/10.1109/REDW.2003.1281316. ISBN 0780381270

[O'Bryan u. a. 2011] O'Bryan, Martha V. ; LaBel, Kenneth A. ; Pellish, Jonathan A. ; Lauenstein, Jean-Marie ; Chen, Dakai ; Marshall, Cheryl J. ; Oldham, Timothy R. ; Kim, Hak S. ; Phan, Anthony M. ; Berg, Melanie D. ; Campola, Michael J. ; Sanders, Anthony B. ; Marshall, Paul W. ; Xapsos, Michael A. ; Heidel, David F. ; Rodbell, Kenneth P. ; Swonger, Jim W. ; Alexander, Don ; Gauthier, Michael ; Gauthier, Brian: Recent Single Event Effects Compendium of Candidate Electronics for NASA Space Systems. In: *Enhancing context awareness with activity recognition and radio fingerprinting* (2011), 1-13. http://dx.doi.org/10.1109/REDW.2010.6062500. – ISSN 97816055871960270525 7

[Park u. a. 2009] Park, Sang-Hoon ; Ha, Seung-Hwan ; Bang, Kwanhu ; Chung, Eui-Young: Design and analysis of flash translation layers for multi-channel NAND flash-based storage devices. In: *Verizon Ad* 55 (2009), Nr. 3. http://dx.doi.org/10.1109/TCE.2009.5278005. – ISSN 00983063

[Park u. Kim 2009] Park, Sang O. ; Kim, Sung: An efficient multimedia file system for NAND flash memory storage. In: *Solid-Projectile Helical Electromagnetic Launcher* 55 (2009), Nr. 1. http://dx.doi.org/10.1109/TCE.2009.4814426. – ISSN 00983063

[Park u. Kim 2011] Park, Sang O. ; Kim, Sung J.: An Efficient Array File System for Multiple Small-Capacity NAND Flash Memories. In: *The Tofu Interconnect* (2011), 569-572. http://dx.doi.org/10.1109/NBiS.2011.94. – ISSN 97814577078962157041 8

[Patterson u. a. 1988] Patterson, David A. ; Gibson, Garth ; Katz, Randy H.: A Case for Redundant Arrays of Inexpensive Disks (RAID). (1988). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.112.9924

[Poizat 2009] Poizat, Marc: *Total Ionizing Dose - Mechanisms and Effects*. 2009

[Qin u. a. 2012] Qin, Zhiwei ; Wang, Yi ; Liu, Duo ; Shao, Zili: Real-Time Flash Translation Layer for NAND Flash Memory Storage Systems. In: *Cover art]* (2012), 35-44. http://dx.doi.org/10.1109/RTAS.2012.27. – ISSN 97814673088301080 1812

[Rao u. a. 2001] Rao, Ji-Shin ; Lee, V.C.S. ; Wu, Jun: Real-time disk scheduling for block-stripping I2O RAID. In: *Variation issues in on-chip optical clock distribution* (2001), 217-224. http://dx.doi.org/10.1109/EMRTS.2001.934036. ISBN 0769512216

[Rosenblum u. a. 1995] Rosenblum, Mendel ; Bugnion, Edouard ; Herrod, Stephen A. ; Witchel, Emmett ; Gupta, Anoop: The impact of architectural trends on operating system performance. In: *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995, S. 285–298

[Wang u. a. 2014] Wang, Yi ; Huang, Min ; Shao, Zili ; Chan, Henry C. B. ; Bathen, Luis Angel D. ; Dutt, Nikil D.: A Reliability-Aware Address Mapping Strategy for NAND Flash Memory Storage Systems. In: *IEEE Transactions on computer-aided design of integrated circuits and systems* (2014), 1623-1631. http://dx.doi.org/10.1109/TCAD.2014.2347929. – ISSN 0278–0070

[Woodhouse 2008] Woodhouse, David: JFFS: The Journalling Flash File System. (2008). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.128.6156

[Youngjae u. a. 2009] Youngjae, K. ; Tauras, B. ; Gupta, A. ; Urgaonkar, B.: FlashSim: A Simulator for NAND Flash-Based Solid-State Drives. In: *Title Page i* (2009). http://dx.doi.org/10.1109/SIMUL.2009.17. ISBN 9781424448630