FERNUNIVERSITÄT IN HAGEN
Fakultät für Mathematik und Informatik
Lehrgebiet Kooperative Systeme - Praktische Informatik VI

Masterarbeit

Dateisysteme für Flashspeicher: Übersicht, Problembereiche und Stand der Entwicklung

vorgelegt am 14. August 2013 von:
Wolfgang Münst
Matrikelnummer xxxxxxx

betreut durch:
Dr. XXX
und
apl. Prof. Dr. XXX

Inhalt

Verzeichnis der Abbildungen

Verzeichnis der Tabellen

Verzeichnis der Abkürzungen

ECC	Error-correcting Code
eCos	Embedded Configurable Operating System
EEPROM	Electrically Erasable Read-Only Memory
EXT3	Third extended file system
F2FS	Flash-Friendly File System
FAT	File Allocation Table
FG	Floating Gate
FTL	Flash Translation Layer
GB	Gigabyte
I/O	Input/Output
JFFS	Journalling Flash File System
KB	Kilobyte
LRU	Least recently used
MB	Megabyte
MLC	Multi-Level Cell
MOS	Metal Oxide Semiconductor
NAND	Not AND (Boolean)
NTFS	New Technology File System
NOR	Not OR (Boolean)
OOB	Out of Band
pSOS	portable Software on Silicon
RAID	Redundant Array of Independent/Inexpensive Disks
ROM	Read-only Memory
SLC	Single-Level Cell
SSD	Solid State Drive
UBI	Unsorted Block Images
UBIFS	Unsorted Block Images File System
XIP	Execute in Place
YAFFS	Yet Another Flash File System

1 Einleitung

Flashspeicher werden bereits seit vielen Jahren eingesetzt. Die grundlegende Technik dazu stammt bereits aus den 1980er Jahren [2]. Ende der 1990er Jahre verbreitete sich die Technik zunehmend, sie wurde vielfach in MP3-Player verbaut und ist seit Mitte der 2000er in praktisch jedem Smartphone integriert. Sie ist aus keinem Tablet-PC wegzudenken und auch in Ultra-Books setzt sie sich durch [3]. Im Gegensatz zur herkömmlichen Festplattentechnik sind Flashspeicher insbesondere für den mobilen Einsatz besser geeignet: sie bieten eine hohe Vibrations- und Stoßfestigkeit, geringes Gewicht, entwickeln kaum Abwärme und haben demzufolge einen deutlich geringeren Energieverbrauch (im Vergleich zu herkömmlichen Festplatten ca. 80% weniger Stromaufnahme im Ruhezustand und ca. 60% weniger bei Zugriffen [4]). Weiterhin bietet er sehr kurze Zugriffszeiten und hat keine Geräuschentwicklung.

Allerdings werden diese Vorteile durch einige Nachteile erkauft. So ist beispielsweise der Preis pro Gigabyte (GB) Speicherplatz noch deutlich höher. Eine herkömmliche Festplatte gibt es derzeit (Juli 2013) ab ca. 0,03€ pro GB (vgl. [5]), während die günstigsten auf Flash-Speicher basierenden Massenspeicher ab ca. 0,55€ pro GB erworben werden können (vgl. [6]).

Diese Arbeit stellt den Stand der Wissenschaft zu einzelnen Komponenten von Flashspeicher und Flashfirmware dar und zeigt, wie die Nutzung des Flashspeichers durch Einsatz von intelligenten Dateisystemen oder Übersetzungsschichten verbessert wurde und welche Herausforderungen bei der weiteren Verbesserung der Softwareschichten bestehen. Dazu werden mögliche Vorgehensweisen der wichtigsten Flash-Komponenten wie Wear Leveling, Garbage Collection und Adressierungsmöglichkeiten vorgestellt. Die Funktionsweise von einzelnen Flash-Dateisystemen wird erläutert und erörtert, für welche Anwendungsszenarien diese sinnvoll eingesetzt werden können.

Die folgenden Kapitel sind folgendermaßen aufgebaut: in Kapitel 2 wird zunächst die Hardware erklärt. Dort werden die technischen Grundlagen erläutert, die zu der sehr unterschiedlichen Verhaltenscharakteristik des Speichers im Vergleich zu anderen

Technologien führt. Kapitel 3 beschäftigt sich allgemein mit Dateisystemen. Es wird zunächst dargestellt, welche unterschiedlichen Arten von Dateisystemen es gibt und wie Dateisysteme auf Flashspeicher zugreifen können. Danach wird in Kapitel 4 erläutert, welche prinzipiellen Mechanismen für Flashspeicher im System implementiert sein müssen, damit dieser wirtschaftlich genutzt werden kann, wie z.B. Garbage Collection, Wear-Leveling oder Error-Correction. Dabei wird auf die verschiedenen Vorschläge der Wissenschaft zur Lösung der Probleme, die durch Flash-Besonderheiten auftreten, eingegangen und der aktuelle Stand beschrieben. Dem folgend werden in Kapitel 5 einige Flash-Dateisysteme erläutert und gezeigt, wie in diesen die Probleme der Flash-Technik gelöst wurden. Kapitel 6 zieht ein kurzes Fazit über den aktuellen Stand.

2 Funktionsweise Flashspeicher

Im Folgenden soll erläutert werden, wie Flashspeicher technisch funktionieren. Erst dadurch werden die Einschränkungen der Technik deutlich, die durch intelligente Controller oder Dateisysteme umgangen werden sollen.

2.1 Grundlegende Technik

Ursprünglichen gab es im Halbleiter-Speicherbereich die ROM[1]-Technik, in der der Inhalt der Speicherzellen bei der Produktion festgelegt wurde und deren Inhalt später nicht mehr verändert werden konnte. 1971 wurde dann von Frohmann-Bentchkowsky ein Floating Gate Transistor vorgestellt, in dem Elektronen gespeichert werden konnten (vgl. [7]). Daraus wurden EPROMs[2] entwickelt, deren Inhalt durch ultraviolettes Licht (UV-Licht) gelöscht werden konnte. Es folgten schließlich EEPROMs[3], die es ermöglichten, den Inhalt der Speicherzellen elektrisch zu löschen und wieder zu beschreiben. Sie wurde Anfang der 1980er durch Toshiba erfunden (vgl. [8]). Diese Technik wird in den heutigen Flashspeichern genutzt und bildet die Basis der USB-Sticks, Handyspeicher und Solid State Drives (SSDs). ROM in den Abkürzungen erscheint mittlerweile fehl am Platz, ist jedoch ein Hinweis auf die Technik, aus der sie selbst hervorgegangen ist.

Als nicht-volatile Speichertechnik wird von Flash eine Vorhaltezeit von mindestens zehn Jahren erwartet, in der die Daten noch lesbar sein müssen (vgl. [9]).

Um den Zustand einer Speicherzelle zu repräsentieren, werden Ladungen verwendet. Die Basis der Flash-Technologie ist der MOS-Transistor[4], der durch ein Floating Gate ergänzt wird, welches zwischen dem Gate und dem Substrat liegt. Die Ladungen werden hierbei auf dem Floating Gate gespeichert, welches zwischen zwei Isolatoren liegt. Durch diese Isolatoren ist sichergestellt, dass die Ladung auch ohne Stromzufuhr mehrere Jahre unverändert erhalten bleibt. Auch Lesevorgänge, also die

[1] ROM = Read-only Memory, Nur-Lese-Speicher
[2] EPROM = Erasable Read-Only Memory, Lösch- und programmierbarer Nur-Lese-Speicher
[3] EEPROM = Electrically Erasable Programmable Read-Only Memory, also elektisch lösch- und programmierbarer Nur-Lese-Speicher
[4] MOS = Metal Oxide Semiconductor, also Metalloxid-Halbleiter

Abfrage des Ladezustands, verändern den Inhalt nicht. Der schematischen Aufbau eines Floating-Gate-Transistors ist in Abbildung 1 dargestellt.

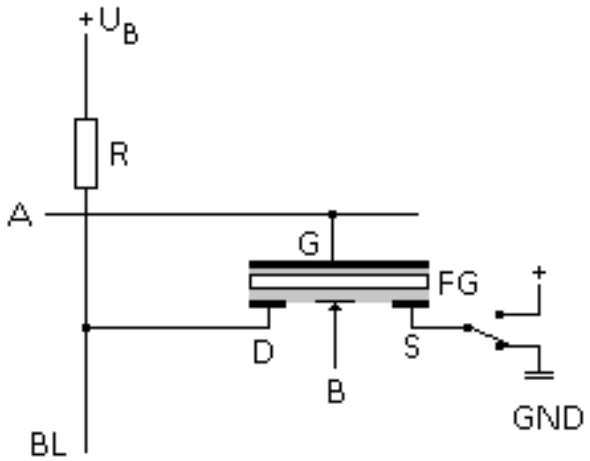


Abbildung 1: Schaltbild Floating Gate Transistor (eigene Darstellung, basierend auf [11])

Der gezeigte Transistor besitzt die Anschlüsse Drain (D), Source (S) und Gate (G). G ist dabei durch eine Isolierschicht (üblicherweise SIO_2, Siliziumdioxid, im Bild grau) vom Bulk (B) getrennt.

Wenn an G gegenüber B keine positive Spannung liegt, kann zwischen D und S kein Strom fließen. In dem Fall kann an der Bitleitung (BL) über den Widerstand R die Betriebsspannung gemessen werden.

Legt man nun an G eine Spannung gegenüber B an, wird dadurch ein sogenannter leitender Kanal erzeugt, so dass zwischen D und S nun Strom fließen kann. In diesem Fall kann an BL nur eine verringerte Spannung gemessen werden.

Diese Beschreibung hat bislang das Floating Gate (FG) außer Betracht gelassen. Es befindet sich ohne eigene, direkte Anschlüsse innerhalb der Isolierschicht: es „schwebt“ darin, daher auch der Name. Wenn nun in dem Floating Gate Elektronen „gespeichert“ werden, die eine negative Ladung haben, wird eine höhere Spannung zwischen G und B benötigt, um den leitenden Kanal zu erzeugen, als es ohne die Elektronen auf dem Floating Gate notwendig wäre. Somit kann eine Spannung zwischen G

und B angelegt werden und an der BL der Widerstand gemessen werden. Je nachdem, ob auf dem Floating Gate Elektronen vorhanden sind, wird sich das Ergebnis unterscheiden. Diesen Effekt wird ausgenutzt, um Zustände zu speichern, z.B. Zustand „1“ wenn eine hohe Spannung gemessen wird (also sich keine Elektronen auf dem FG befinden) und Zustand „0“ wenn eine geringere Spannung gelesen wird. Im Gegensatz zu anderen Technologien (z.B. SDRAM-Chips) handelt es sich bei Flashspeicherzellen um einen nicht-zerstörenden Lesevorgang, d.h. der Zustand der Zelle wird durch Auslesen nicht verändert (vgl. [10]).

Dieser Zustand muss auch eingestellt werden können, dabei spricht man von der Programmierung der Flash-Zelle (vgl. [11]). Üblicherweise werden die Elektronen entweder per „Heiße Elektronen“-Methode oder per „Fowler-Nordheim-Tunneling“ auf das Floating Gate gebracht und wieder entfernt (vgl. [9], [12]).

Die „Heiße Elektronen“-Methode funktioniert so, dass ein elektrisches Feld erzeugt wird, durch das den Elektronen mehr Energie zugeführt wird, als sie durch ihre Gitterschwingung (engl. „lattice vibration“) wieder abgeben. Dadurch bewegen sie sich immer schneller und haben eine Chance, den Isolator zu durchbrechen und im Floating Gate zu landen (vgl. [12]). Das „Fowler-Nordheim-Tunneling“ ist ein quantenmechanischer Effekt und macht sich zu nutze, dass durch Anlegen einer hohen Spannung einzelne Elektronen in das Floating Gate durch die Isolierschicht getunnelt werden können (vgl. [12]).

Die Flash-Zellen werden gelöscht, indem auf den Anschluss S eine hohe positive Spannung und gleichzeitig das Gate G auf Massepotenzial 0V angelegt wird. Dies geschieht indem der Schalter am Anschluss S von GND auf die positive Spannung umgelegt wird. Die Elektronen des Floating Gates fließen dann über S ab. Bei diesem Vorgang bleiben allerdings einzelne Elektronen immer wieder in der Isolierschicht „hängen“ und schädigen den Isolator dauerhaft (vgl. [9], [11]). Dies führt zu einem zunehmenden Abfluss von Ladungsträgern und erschwert dadurch das korrekte Auslesen der Zelle.

Der oben erwähnte Schalter am Anschluss S wird in der Praxis nicht für jede einzelne Flash-Zelle realisiert, sondern wird von vielen Flash-Zellen gleichzeitig genutzt.

Man spricht von einem Flash-Block. Das führt jedoch dazu, dass solche Zellen nicht einzeln gelöscht werden können, sondern bei einem Löschvorgang immer der ganze Block gelöscht wird. Daher auch der Name „Flash“[5], denn es kann ein ganzer Block „wie der Blitz“ gelöscht werden.

2.2 SLC und MLC

Wird bei einer Flash-Zelle nur unterschieden, ob das Floating Gate Elektronen enthält oder nicht, spricht man von einer Single-Level Cell (SLC). Beim ersten Lesevorgang des Zustands einer SLC kann sofort der Zustand ermittelt werden.

Es ist technisch auch möglich, unterschiedliche Ladungsträgerdichten auf das Floating Gate zu bringen. Wenn statt bisher zwei Zuständen etwa vier Zustände gespeichert werden sollen, kann das durch „keine Elektronen“, „wenige Elektronen“, „mittel viele Elektronen“ und „viele Elektronen“ auf dem FG repräsentiert werden. Eine solche Zelle wird Multi-Level Cell (MLC) genannt.

Dies führt allerdings zu einigen Einschränkungen: Zum einen dauert der Lesevorgang des Zustandes bis zu vier mal länger als bei SLC, da die einzelnen Spannungsbereiche nacheinander durchprobiert werden müssen, bis der Transistor schaltet. Zum anderen steigt die Fehlerrate, da die Menge der Elektronen, die auf das FG gebracht werden müssen, nun viel exakter gesteuert werden muss, als es bei SLC der Fall war (vgl. [10]). Gerade weil die Flash-Zellen selbst bei jedem Löschvorgang „altern“, lässt die für mehrere Zustände notwendige Lesegenauigkeit mit der Zeit nach und führt dazu, dass MLCs das Ende ihrer Lebensdauer deutlich schneller erreichen, als es bei SLCs der Fall ist (vgl. [13]).

MLCs können mit noch mehr als zwei Bits pro Zelle genutzt werden. Werden drei Bits pro Zelle gespeichert, spricht man von Triple-Level-Cells (TLC). Diese Technik wird bereits seit 2011 für manche Flash-Massenspeicher eingesetzt (vgl. [14]). Für drei Bits müssen allerdings schon acht unterschiedliche Spannungsbereiche unterschieden werden, für vier Bits 16.

2.3 NOR und NAND

Die Floating Gate Transistoren können auf unterschiedliche Weise miteinander verbunden sein: NOR oder NAND. Die Transistoren sind so eng wie es die aktuelle Schalt-

[5] engl. „flash“ = Blitz

technik und Störsicherheit erlaubt auf rechteckigen Flächen zusammengefasst: momentan wird 10nm-20nm Produktionstechnik bei Toshiba und Samsung eingesetzt (vgl. [15], [16]), die beide auch Flashspeicher-Hersteller sind. Über die Schalttechnik kann jede einzelne Zelle angesprochen werden. Die horizontalen Verbindungen werden „Wortleitungen“ („wordlines“) genannt, in der Abbildung 1 entspricht das der Leitung A. Die vertikalen Verbindungen werden als „Bitleitungen“ („bitlines“) bezeichnet, in Abbildung 1 ist das die Leitung BL.

Die Wortleitungen werden benutzt, um die Zellen auszuwählen, auf die eine Lese- oder Schreiboperation ausgeführt werden soll. Die Bitleitungen verbinden die Zellen mit den jeweiligen Lese- oder Schreibschaltkreisen.

Das Schaltbild für Transistoren, die zu NOR-Flash zusammengefasst werden, ist in Abbildung 2 dargestellt.

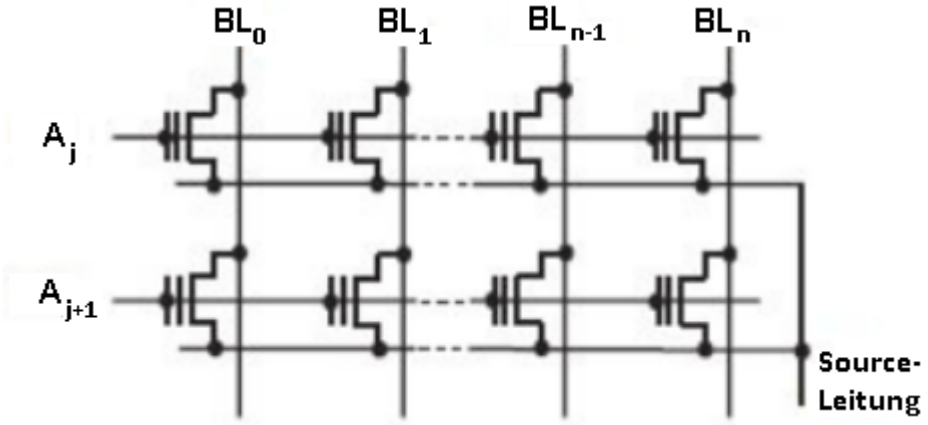


Abbildung 2: NOR-Schaltung (vgl. [10])

Hier kann man erkennen, dass jede Zelle direkt einzeln ausgewählt und angesprochen werden kann. Auf NOR-Flash kann daher wahlfrei zugegriffen werden und es werden 100% zuverlässige Bits garantiert, die Fehlerkorrekturmechanismen überflüssig werden lassen (vgl. [10], Kap. 2.1.3.1).

Es ist auch eine Zusammenschaltung in NAND-Form möglich, wie in Abbildung 3 gezeigt.

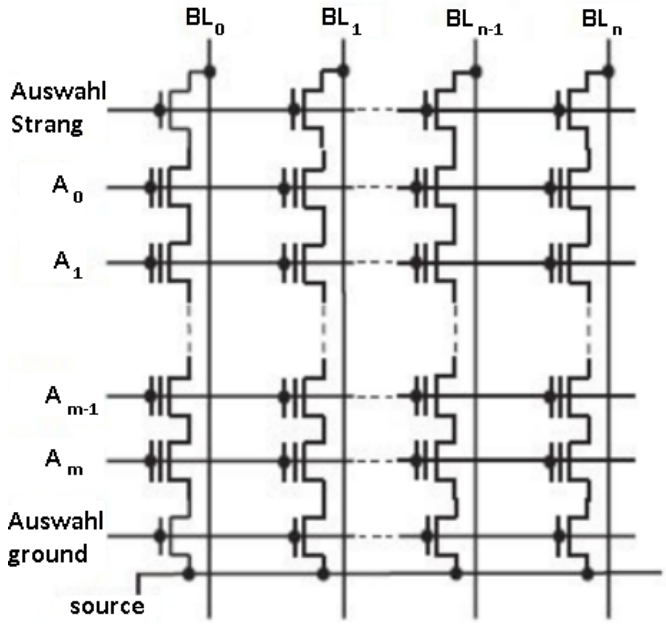


Abbildung 3: NAND-Schaltung (vgl. [10])

Hier sieht man, dass mehrere Transistoren pro Bitleitung zu einem Strang/Kette („string“) zusammengeschlossen sind und sich jeweils einen Source- und Ground-Anschluss teilen. Üblicherweise werden so 8, 16 oder 32 Zellen zusammengeschaltet (vgl. [10]). Dadurch kann auf die Zellen einer Bitleitung nur gemeinsam zugegriffen werden.

Daraus folgt, dass auf NAND-Speicher im Gegensatz zu NOR-Speicher kein Execute In Place (XIP) möglich ist. Es fehlt dort die wahlfreie, bitweise Zugriffsmöglichkeit. Execute in Place bedeutet, dass Programmcode direkt ausgeführt werden kann, ohne zunächst in den Hauptspeicher des Systems geladen werden zu müssen (vgl. [17]). Code, der direkt ausgeführt werden soll, erfordert weiterhin 100% Zuverlässigkeit des Speichermediums, da ansonsten im Fehlerfall ein Systemfehler verursacht wird. NOR-Speicher garantiert das, NAND jedoch nicht.

Zusammenfassend hier die Unterschiede von NOR- und NAND-Flashspeicher:

	NOR	***NAND***
Geschwindigkeit (Löschen eines Blocks)	Langsam: ca. 0.7-1s [1], [18], [19], [20]	Schnell: ca. 0.5-2 ms [18], [19], [20], [22]
Geschwindigkeit (Schreiben)	Langsam: ca. 0.25 MB/s [19], [21]	Schnell ca. 7.5 MB/s [19], [21]
Geschwindigkeit (Lesen)	Sehr schnell: ca. 30-60 MB/s [19]	Schnell: ca. 25-40 MB/s [19]
Löschzyklen	10.000 – 100.000	SLC übersteht deutlich mehr Löschzyklen als MLC 10.000 – 1.000.000 [1], [21], [23], [24]
Zugriffsart	Zufällig	Sequenziell
Execute in Place (XIP)	Ja	Nein (aber theoretisch durch Anpassungen möglich) [17]
Platzbedarf	Hoch	Niedrig (ca. 40% von NOR) [11]
Kosten	Hoch	Niedrig

Tabelle 1: Eigenschaften von NOR- und NAND-Flashspeicherbausteinen

Bei den Angaben zur Geschwindigkeit ist noch zu beachten, dass diese durch Parallelisierung mehrerer Flashbausteine deutlich gesteigert werden kann. Dies wird beispielsweise in Solid State Drives intern umgesetzt, indem der SSD-Controller auf mehrere Flashbausteine parallel zugreift.

2.4 Zuverlässigkeit und Fehlerquellen

Im Flashspeicher kann es durch verschiedene Ursachen zu fehlerhaften Daten kommen. Hier sollen einige Fehlerquellen vorgestellt werden.

2.4.1 Nicht mehr nutzbare Blöcke (Wear Out)

Zu fehlerhaften Blöcken kann es einerseits bereits bei der Produktion von NAND-Flash kommen. Bei SLC-Zellen sind bis zu 2% aller Blöcke bei der Auslieferung bereits defekt, bei MLC-Zellen können es bis zu 5% sein (vgl. [25], S. 8). Zusätzlich sind auch fehlerfreie Flashblöcke nur begrenzt oft löschbar, bevor sie unzuverlässig werden. Das liegt daran, dass sich in der Oxidschicht um das Floating Gate mit jedem Löschvorgang zusätzliche Elektronen ansammeln und dort gefangen bleiben. Irgendwann werden es dann so viele sein, dass der Transistor nicht mehr schalten kann, und diese Zelle dann immer in dem gleichen Zustand ausgelesen wird und somit zur Datenspeicherung wertlos wird.

Ob solch ein Fehler vorliegt, kann festgestellt werden, indem ein Block gelöscht wird und direkt danach alle Zellen auf ihren erwarteten Wert überprüft werden.

2.4.2 Dauerhaftigkeit der Speicherung (Data Retention)

Hierbei treten Mängel auf, wenn sich die Ladung des Floating Gate Transistors durch Leckströme über die Zeit von selbst verändert (vgl. [26], Folie 21 und [25], S. 8). Wenn dabei der Schwellwert überschritten wird, bei dem ein Bit als 0 oder 1 interpretiert wird, kann ein falscher Wert ausgelesen werden. Dieser Effekt wird stärker, je öfter ein Block bereits gelöscht wurde (vgl. [26], Folie 22).

Der Fehler ist jedoch nur vorübergehender Natur und kann behoben werden, indem der betroffenen Block gelöscht und neu geschrieben wird.

2.4.3 Durch Lesen verursachte Fehler (Read Disturb Error)

Während Seiten im NAND-Flash gelesen werden, können benachbarte Zellen im gleichen Block durch Störsignale beeinflusst werden, wodurch sich Elektronen auf einigen Floating Gates von diesen benachbarten Zellen einlagern könnten (vgl. [26], Folie 19). Dies ist besonders bei MLC problematisch, da dort geringere Ladungsänderungen nötig sind, um bereits zur Änderung logischer Bits zu führen.[6] Als Abhilfe wird empfohlen, nach einer bestimmten Anzahl von Lesezugriffen die Daten des Blocks neu zu schreiben. Bei SLC-Speicher sollte dies circa alle 1.000.000sten Lesezugriff und bei MLC-Speicher etwa jeden 100.000sten Lesezugriff geschehen (vgl. [26], Folie 20).

Auch dieser Fehler führt zu keinem dauerhaften Problem und kann per ECC effektiv erkannt werden, so dass die Daten in solch einem Fall direkt neu geschrieben werden sollten.

2.4.4 Durch Schreiben verursachte Fehler (Write Disturb Error)

Ähnlich wie der Read Disturb Error können auch beim programmieren von Zellen umliegende Zellen des gleichen Blocks beeinflusst werden und einige Elektronen auf die Floating Gates benachbarter Zellen gelangen. Auch dieser Fehler ist nicht dauerhaft und kann durch ECC erkannt werden. Dann sollten die Daten des Blocks gelöscht und neu geschrieben werden.

[6] Deswegen werden sie teilweise auch „bit flip error“ genannt

2.5 Zugriff

Für den Zugriff auf NOR- und NAND-Flashspeicher gibt es Unterschiede bei den Lese- und Schreibzugriffen.

2.5.1 Lesen und Schreiben

NOR-Flash kann bitweise gelesen und geschrieben werden. Es gilt jedoch weiterhin, dass nicht **Über**schrieben werden kann. Wenn ein Bit neu gesetzt werden soll und sich bereits Elektronen auf dem Floating Gate dieses Bits befinden, muss der vorherige Wert zuerst gelöscht werden. Das ist allerdings nur blockweise möglich, wie im nächsten Unterabschnitt 2.5.2 beschrieben werden wird.

NAND-Flash hingegen erlaubt nur das Lesen und Schreiben einer gesamten Seite. Sein Aufbau ist in Abbildung 4 dargestellt. Eine Seite sind mehrere zusammengefasste Bits, die sich einen gemeinsamen zusätzlichen Speicherbereich namens „Out of Band“ (OOB) teilen, in dem Verwaltungsinformationen gespeichert sind (z.B. Prüfsummen und Marker).

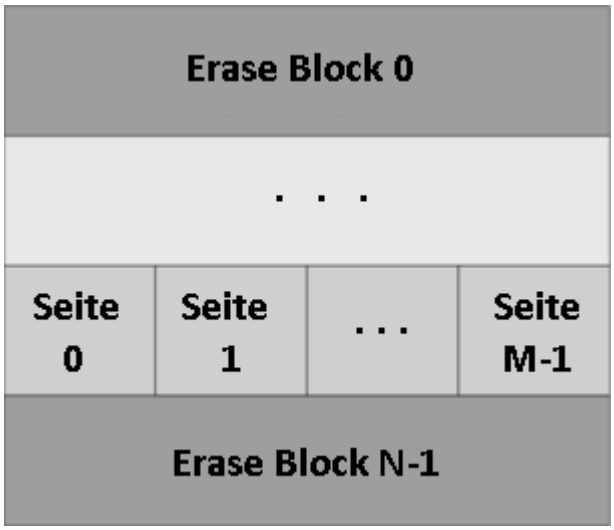


Abbildung 4: Aufbau NAND-Flash (vgl. [90], Figure 2.2)

Ein NAND-Flashbaustein besteht aus vielen Blöcken, die wiederum aus vielen einzelnen Flash-Seiten bestehen. Die Größenordnung für eine Seite liegen aktuell im Bereich von 2 KB plus 64 Bytes für den OOB-Bereich. Dieser Wert hat der Zeit zugenommen, Anfang 2000 war die übliche Aufteilung noch 512 Byte plus 16 Byte für OOB. Ein Block fasst in der Regel 64 bis 128 Seiten zusammen. Geschrieben wird NAND-Speicher, indem die Daten in einen internen Buffer abgelegt werden und dann der „Schrei-

ben“-Befehl aufgerufen wird. Innerhalb einer Seite kann bis zu zehn Mal geschrieben werden, bevor der Inhalt dieser Seite in einen undefinierten Zustand übergeht und erst nach einem Löschvorgang auf den Block wieder verwendet werden kann. Die kleineren Seitenabschnitte werden „Subpage“ (dt. Unter-Seite) genannt. Dies wird jedoch nicht von allen Flashspeichern unterstützt, bei manchen können nur ganze Seiten geschrieben werden.

2.5.2 Löschen

Gelöscht werden kann Flashspeicher nur blockweise, dies gilt sowohl für NOR als auch NAND-Bausteine.

Es ist eine Aufteilung in beliebig kleine Blöcke möglich, die voneinander unabhängig gelöscht werden können; es wäre sogar technisch möglich, dies z.B. auf die Größe eines Bytes zu reduzieren. Je kleiner jedoch die Löscheinheit festgelegt wird, desto mehr Transistoren und Platz benötigten die erforderlichen Schaltungen (vgl. [10]). Dadurch steigen die Kosten. Es muss daher eine Abwägung zwischen Kosten und Vorteilen der kleineren Löscheinheiten gefunden werden. Die Größenaufteilung der Blöcke und Seiten ist durch die Hardware festgelegt und kann nicht durch Software verändert werden.

Mögliche Aufteilungen für Flashspeicher sind in der folgenden Abbildung 5 dargestellt.

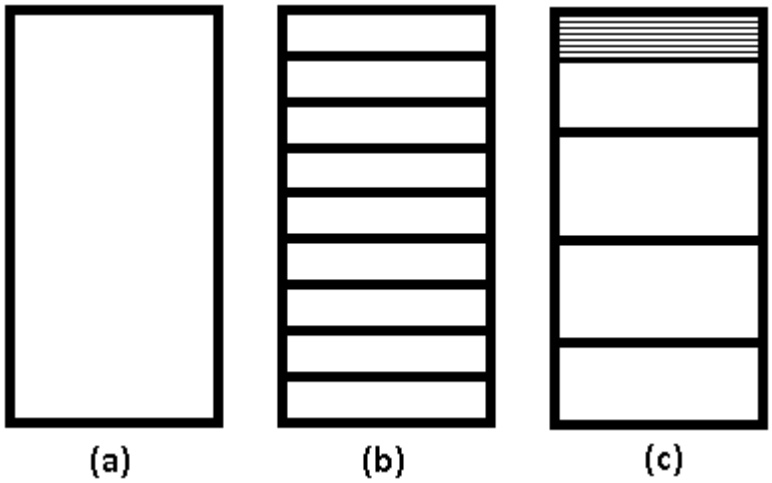


Abbildung 5: Mögliche Aufteilung von Lösch-Blöcken

Abbildung 5 (a) zeigt einen Speicher, der aus nur einem einzigen Lösch-Block besteht. Dies war z.B. bei früheren EPROMs der Fall, die nur gesamt per UV-Licht gelöscht werden konnten. Die in Abbildung 5 (b) gezeigte symmetrische Aufteilung ist heute der Standard bei NAND-Speichern, z.B. in SSDs oder USB-Sticks. Dort wird der gesamte Speicherplatz in gleich große Einheiten unterteilt, die voneinander unabhängig gelöscht werden können. Eine weitere Möglichkeit wäre die in Abb. 5 (c) dargestellte, asymmetrische Variante: dort sind manche Blöcke größer als andere. Dies ist z.B. dann sinnvoll, um Daten in kleinere Blöcke zu speichern, da diese sich öfter ändern. Code wird selten geändert und kann daher auch in größeren Blöcken gespeichert werden, da die Blöcke trotzdem selten gelöscht werden müssen (vgl. [10], Kap. 2.1.5.1). Diese asymmetrische Variante ist hauptsächlich für eingebettete Systeme interessant gewesen, wenn das System dem Zugriffsverhalten entsprechend optimiert wurde.

3 Flashspeicher und Dateisysteme

In diesem Kapitel soll zunächst kurz erläutert werden, welche unterschiedlichen Arten von Dateisystemen es gibt.

3.1 Grundsätzliche Dateisystemarten

Damit Programmierer und Anwender sich nicht selbst mit den Details jedes Systems beschäftigen müssen, auf dem Daten gespeichert werden können, wird von Betriebssystemen ein Dateisystem zur Verfügung gestellt. Eine Datei ist dabei eine Menge von logisch zusammengehörigen Daten, die permanent gespeichert werden kann (vgl. [27], Kap. 10.7.1). Ein Dateisystem ist die Menge aller Dateien, Verzeichnisse und Hilfsdaten, die auf einem Datenträger gespeichert sind (vgl. [28], Kap. 5.3.1).

Über die Jahre wurden verschiedene Ansätze entwickelt, wie ein Dateisystem aufgebaut und organisiert werden kann. Es wurden drei grundlegende Realisierungsmöglichkeiten entwickelt, wie Daten verwaltet werden: In-Place, Journaling und Log-Based. Die folgenden Abschnitte sollen einen kurzen Überblick über diese verschiedenen Dateisystem-Arten geben, damit später gezeigt werden kann, welche Auswirkungen sich daraus für die Leistungsfähigkeit auf unterschiedlicher Hardware ergeben.

3.1.1 In-Place

Dateisysteme, die nach dem In-Place-Verfahren arbeiten, schreiben bei Änderungen sofort über die vorhandenen, zu verändernden Daten. Wenn Daten von Dateiblöcken verändert werden, hat das den Vorteil, dass die Daten am gleichen physikalischen Ort auffindbar sind.

Nachteilig ist jedoch, dass daher veränderte Daten an einer bestimmten Position geschrieben werden müssen und bei herkömmlichen Festplatten gewartet werden muss, bis der Schreib-/Lesekopf dort positioniert ist. Weiterhin kann es beispielsweise bei einem Stromausfall passieren, dass Teile der Daten eines Schreibvorgangs geschrieben wurden, andere aber noch nicht. So kann es zu Inkonsistenzen im Dateisystem kommen, die zu Datenverlusten führen können.

Beispiele für ist In-Place-Dateisysteme sind second extended file system (ext2) oder File Allocation Table (FAT).

3.1.2 Journaling

Journaling-Dateisysteme schreiben ihre Daten genauso wie die In-Place Dateisysteme auf dem Speichermedium. Allerdings wird vor jedem Schreibvorgang in einer Art Logfile (dem „Journal“, auch „write-ahead log“ genannt, vgl. [29]) die beabsichtigte Änderung eingetragen. Durch dieses Journal ist es möglich, auch bei unvorhergesehenen Ereignissen, wie beispielsweise einem Stromausfall oder Systemabsturz, die Konsistenz des Dateisystems schnell wieder herzustellen. So kann überprüft werden, an welcher Stelle eines Schreibvorgangs der Abbruch erfolgte, um an dieser Stelle fortzufahren. In-Place Dateisysteme ohne Journal müssen in solchen Fällen den gesamten Inhalt des Dateisystems lesen, um es auf möglicherweise aufgetretene Inkonsistenzen zu überprüfen. Da die Speicherkapazität über die Jahre stark zugenommen hat, kann ein solcher Scan-Vorgang heutzutage mehrere Stunden dauern.
Allerdings besteht auch trotz Journal eine Chance, dass das Dateisystem bei einem Systemausfall korrupt wird. Festplatten-Controller können aus Performance-Gründen die erhaltenen Schreibbefehle neu anordnen. So kann es passieren, dass Daten der Dateien verändert werden, bevor die Informationen im Journal geschrieben wurden. Dies kann z.B. bei ext3/ext4 aufgrund der Standard-Einstellung des Dateisystems in den meisten Linux-Distributionen vorkommen (vgl. [29], [30]).

Für Flash-Speicher hat dieser Aufbau einen entscheidenden Nachteil: durch das Journal werden pro Speichervorgang auf dem Speicher in Wirklichkeit zwei Schreibvorgänge benötigt: einer im Journal und einer in der eigentlichen Datei. Das erweist sich als Nachteil, da Flash-Blöcke nur eine endliche Anzahl an Schreib-/Löschzyklen überstehen und somit die Haltbarkeit der Flash-Hardware durch Journaling-Dateisysteme reduziert wird.

3.1.3 Log-Structured

Bei Log-basierten Dateisystemen („log-structured file system“) **ist** das Journal das Dateisystem. Hier werden Datenänderungen nicht mehr zusätzlich zum Journal-Eintrag geschrieben, sondern es lässt sich direkt aus dem Journal der komplette Inhalt des Massenspeichers rekonstruieren. Das Journal besteht aus vielen gleich großen Segmenten und alle Änderungen werden am Ende des Logs angefügt. Sollen Daten geändert oder überschrieben werden, werden die neuen Daten in einem neuen Eintrag des

Journals geschrieben und der bisherige Eintrag als ungültig markiert, wie in Abbildung 6 gezeigt.

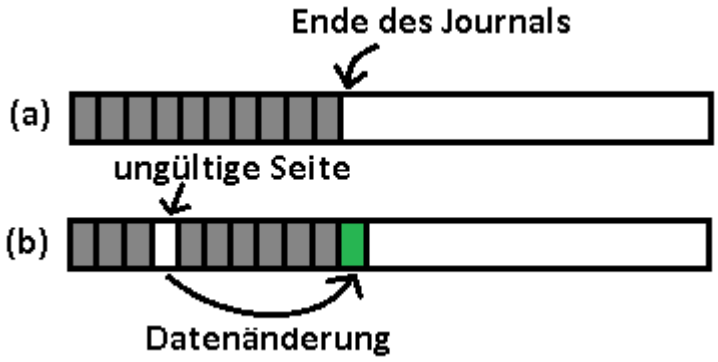


Abbildung 6: Datenänderung bei Log-basierten Dateisystemen

Diese Dateisystemart wurde erstmals 1989 von Ousterhout und Douglis vorgeschlagen [31], um die I/O-Geschwindigkeit gegenüber den bisherigen Dateisystemen zu steigern. Die Sorge damals war, dass durch immer schneller werdende Prozessoren, aber kaum verbesserte Zugriffsgeschwindigkeiten der Festplatten sich letztere als Flaschenhals erweisen könnten. Der Grund dafür ist, dass zwar z.B. durch Parallelisierung von mehreren Festplatten die Übertragungsbandbreite leicht erhöht werden kann. Allerdings wurde gezeigt, dass die meisten Zugriffe auf kleine, auf der Platte verstreute Dateien stattfand und die Festplatten-Schreib-/Leseköpfe damals zu 90-95% mit der Suche nach der richtigen Position über der Platte beschäftigt waren (sog. „seek time“, also Suchzeit, vgl. [32], Abschnitt 2.2 und 2.3).

In dieser ersten Veröffentlichung (vgl. [31]) wurde von einer 1000-fachen Verbesserung der I/O-Zugriffe durch Log-basierte Dateisysteme ausgegangen. Später wurde gezeigt, dass dies nur in speziellen Anwendungsfällen, z.B. bei sehr häufiger Metadaten-Änderung, tatsächlich der Fall ist (vgl. [33]). Die erste Implementierung des Konzepts geschah 1992 durch „Sprite LFS“ von Ousterhout und Rosenblum (vgl. [32]). Die große Performance-Steigerung soll durch Pufferung von Schreibzugriffen erreicht werden, die dann mit nur einer Schreiboperation durchgeführt wird. Dadurch wird seltener nach einer neuen Position auf der Festplatte gesucht, die Suchzeit der Festplattenköpfe also stark verringert.

Es treten durch diese neue Design allerdings neue Probleme auf (vgl. [32]): wenn alle neuen Daten und Datenänderungen im Journal angehängt werden, ist die Festplatte

nach einiger Zeit voll. Im Journal gibt es dann viele Datensätze, die ungültig sind, da sie bereits wieder überschrieben oder gelöscht wurden und somit die aktuellen Daten weiter hinten abgelegt sind. Es gibt zwei Möglichkeiten zur Lösung des Problems: einerseits könnte das Journal die ungültigen Einträge mit neuen Journaleinträgen nach und nach überschreiben, andererseits könnte ein Säuberungs-Mechanismus („cleaner“) implementiert werden, der dafür sorgt, dass der verfügbare Speicherplatz möglichst am Stück verfügbar ist. Der erste Vorschlag alleine würden den Geschwindigkeitsvorteil von Log-basierten Dateisystemen wieder zunichte machen, da es durch die starke Fragmentierung der Daten wieder zu häufigen Suchzeiten der Festplattenköpfe käme. Daher wurde ein kombinierter Ansatz aus beidem gewählt: der Speicherplatz wird in Segmente unterteilt (das Journal überschreibt dann immer nur ganze, komplett ungültige Segmente) und der Säuberungs-Mechanismus gibt nach dem Wegsichern der gültigen Daten aus einem Segment immer ein solches wieder ganz frei. Spätere Forschungen haben ergeben, dass dieser Cleaner die Leistungsfähigkeit der Festplatten teilweise sehr stark reduziert und Verbesserungen für den Algorithmus vorgeschlagen (weitere Informationen dazu in [33], [34]).

Hier werden bereits große Ähnlichkeiten zur internen Funktionsweise von Flashspeichern sichtbar, in dem vorhandene Daten technisch bedingt auch nicht überschreibbar sind. Die Segmente werden bei Flashspeicher als „Blöcke“ bezeichnet, der Cleaner wird dort „Garbage Collector“ genannt.

3.2 Architektur für Flashspeichersysteme

Anwendungen greifen auf das Dateisystem zu, das vom Betriebssystem zur Verfügung gestellt wird. Damit es aber selbst die Daten auf das physikalische Speichermedium schreiben kann, sind Zwischenschichten notwendig. Zum einen muss eine Adressübersetzung zwischen der logischen und physikalischen Adresse stattfinden, da Flash-Blöcke aus Performance-Gründen nicht direkt überschrieben werden und daher häufig ihren physikalischen Speicherort verändern. Dies wird Adress Translation genannt. Zusätzlich müssen noch weitere Funktionen durchgeführt werden, wie die Garbage Collection und das Wear Leveling.

Prinzipiell gibt es für Flash-Speicher zwei grundlegende Möglichkeiten und eine Hybrid-Version, wie das geschehen kann:

1. Es kann ein herkömmliches Dateisystem genutzt werden, z.B. NTFS, FAT oder ext3. Unter diesen Dateisystemen muss dann jedoch eine weitere Zwischenschicht eingefügt werden, die sich wie herkömmliche Festplatten verhält und eine Blockzugriffsweise simuliert. Diese Schicht wird „Flash Translation Layer“ (kurz: FTL) genannt und erledigt genau diese Aufgabe: sie nimmt vom Dateisystem Blockzugriff-Befehle entgegen und setzt diese in die notwendigen Befehle um, um die gewünschten Daten auf Flashspeicher zu schreiben oder zu lesen. Neben dieser Adressübersetzung werden auch weitere notwendige Aufgaben für Flashspeicher übernommen, also Garbage Collection und Wear Leveling. Es wird also die Existenz von Flashspeicher vor dem Dateisystem versteckt. Die FTL kann dabei entweder im Betriebssystem (vgl. Abb. 7) oder als Firmware in der Hardware implementiert sein (z.B. in SSDs, vgl. Abb. 8). Gerade bei Hardwareimplementierungen gelten die genutzten Algorithmen jedoch als Firmengeheimnisse, daher ist nur wenig über die interne Funktionsweise der meisten Hardware-FTLs bekannt.
2. Eine andere Möglichkeit ist, das Dateisystem diese Umsetzung selbst vornehmen zu lassen. Dann muss es jedoch auch dafür ausgelegt sein und muss alle für Flash notwendigen Operationen und Aufgaben selbst durchführen. Solche Dateisysteme werden Flash-Dateisysteme genannt (vgl. Abb. 7/ rechts, die Flash-Dateisysteme JFFS, YAFFS).
3. Es gibt neuerdings auch einen Hybridansatz zwischen den ersten beiden Optionen, bei dem Aufgaben durch das Flash-Dateisystem ausführt werden, andere jedoch auf die Hardware-Firmware verlagert sind. Ein Beispiel dafür ist das Flash-Friendly File System (F2FS, siehe Kap. 5.5), welches z.B. die Garbage Collection selbst durchführt, jedoch das Wear Leveling der Firmware überlässt (vgl. Abb. 8 /rechts, F2FS).

Um alle Aspekte von Flash durch das Flashdateisystem steuern zu können, muss dem Betriebssystem allerdings der direkte Zugriff auf die Flash-Hardware möglich sein, ohne dass die Interna hinter einer Firmware versteckt sind.

In Abbildung 7 ist dieser schematische Aufbau für den Zugriff auf Flashspeicher ohne eigene Firmware beispielhaft dargestellt. Die einzeln dargestellten Aufgaben, beispielsweise „Bad Block Management“, können (aber müssen nicht!) in jeder der angegebenen Schichten implementiert oder aktiviert sein.

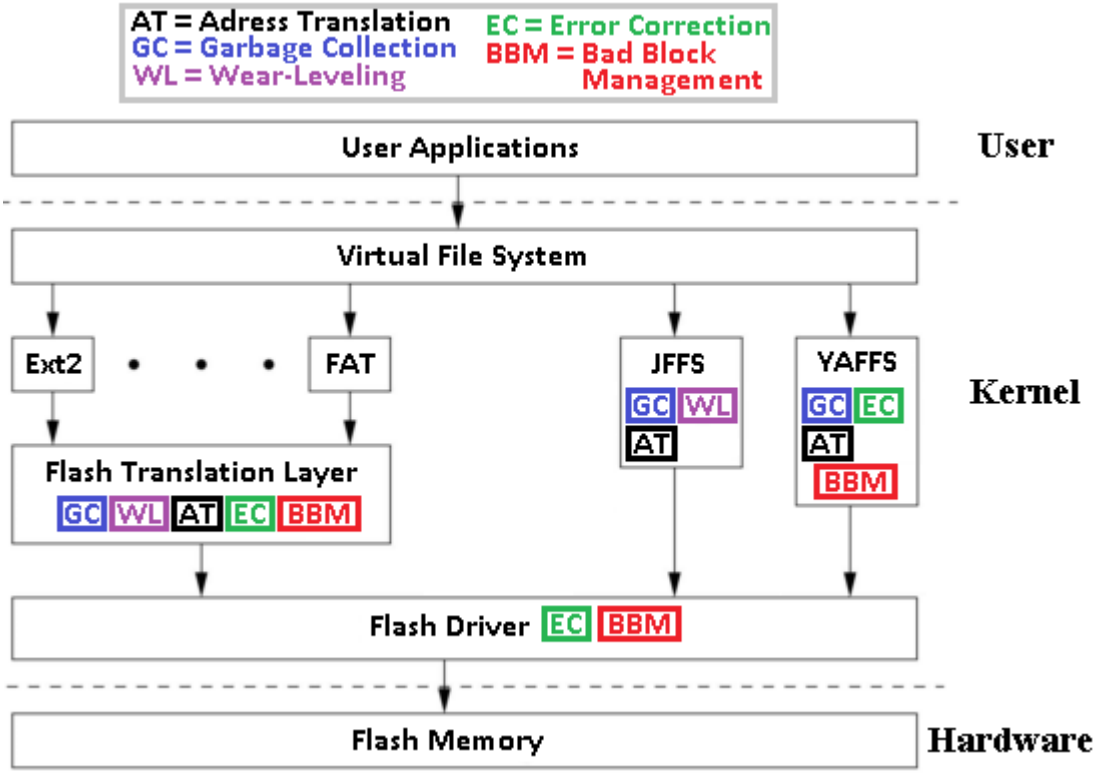


Abbildung 7: Flash-Ebenen zwischen Anwendung und Hardware bei Firmware-FTL (basierend auf [89] / S. 907, Abb. 1)

Es gibt Flash-Geräte, die einen eigenen Controller mit eigener Firmware mitbringen und über einen Blockschnittstelle angesprochen werden können. In diesen sind die Flash-Besonderheiten hinter der Firmware verborgen. Der schematische Aufbau verändert sich dadurch ein wenig, wie in Abbildung 8 dargestellt ist.

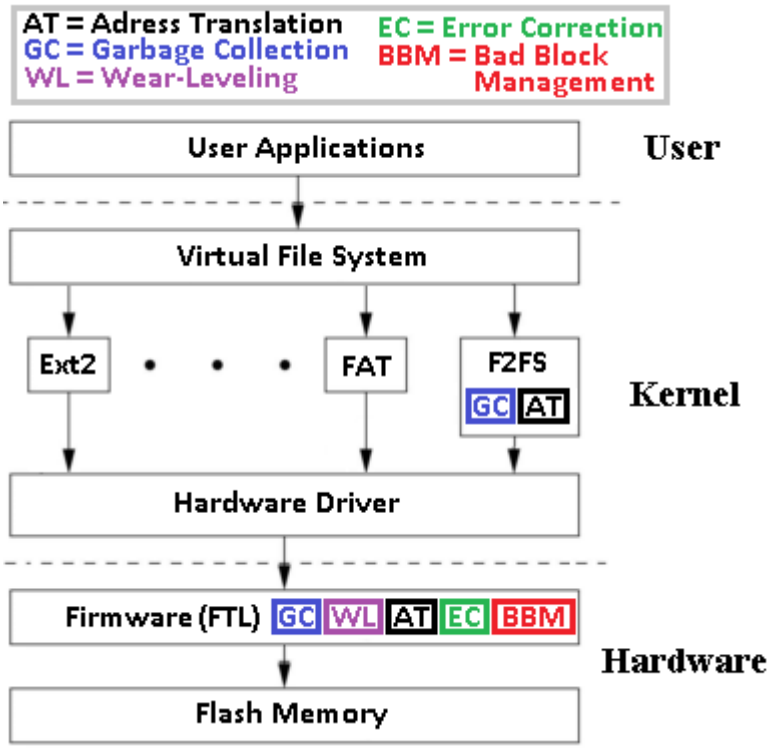


Abbildung 8: Flash-Ebenen zwischen Anwendung und Hardware bei Firmware-FTL (basierend auf [89] / S. 907, Abb. 1)

Beide Ansätze bieten einige Vor- und Nachteile. Eine in die Hardware integrierte FTL ermöglicht hohe Kompatibilität mit bestehenden Systemen. Es ist dadurch sehr einfach, beispielsweise alte Festplattentechnik durch neue Solid State Drives zu ersetzen, da jedes Betriebssystem auf Endkundengeräten sie erkennt und sofort einsetzen kann. Auf der anderen Seite ist dadurch jedoch der Weg verbaut, die meisten Flash-Dateisysteme zu nutzen, da diese nicht funktionieren, wenn der Controller im Flashspeicher in einer tieferliegenden Schicht Verwaltungsaufgaben erfüllt, die bereits das Dateisystem durchführt.

Der direkte Zugriff auf die Flashhardware hat den Vorteil, dass einige der Verwaltungsaufgaben effizienter durchgeführt werden können, da bereits im Dateisystem auf sie Rücksicht genommen werden kann.

4 Problemfelder und Algorithmen

Um Flashspeicher sinnvoll einsetzen zu können, müssen Management-Funktionen auf ihm ausgeführt werden. Hier werden die dazu notwendigen Mechanismen vorgestellt.

4.1 Garbage Collection

4.1.1 Allgemeines

Flashspeicher hat gegenüber anderen Speicherarten das Problem, dass Daten nicht direkt überschrieben werden können. Bei Datenänderungen wäre es daher notwendig, dass der gesamte Block, in dem zu ändernde Daten liegen, gelöscht wird, um danach die veränderten Daten neu zu schreiben – und das selbst dann, wenn nur eine Flash-Seite im gesamten Block geändert würde. Da Flash-Blöcke nur begrenzt oft gelöscht werden können, bevor sie nicht mehr zuverlässig funktionieren und jeder Löschvorgang (relativ gesehen) lange dauert, will man unbedingt vermeiden, dass bei jeder Datenänderung die Daten gelöscht und neu geschrieben werden.

Dies wird umgangen, indem die neu zu schreibenden Daten in eine neue, freie Seite geschrieben werden und die ursprüngliche Seite als ungültig markiert wird.

Dies führt jedoch zu dem Problem, dass der freie Speicherplatz ausgehen wird. An dieser Stelle gibt es auf der Flash-Speicher viele Blöcke, die ungültige und gültige Seiten gleichzeitig beinhalten. Dazu ist ein Vorgang notwendig, der ungültige Seiten wieder als freien Speicherplatz zur Verfügung stellt: dieser Mechanismus wird Garbage Collection genannt. Um wieder Speicherplatz zur Verfügung zu haben, müssen Blöcke gelöscht werden, die ungültige Seiten enthalten. Das geschieht wie in Abb. 9 gezeigt. Zuerst muss ein Block zur Bereinigung gewählt werden (1), der sogenannte victim block[7]. Die gültigen Seiten des Blockes werden davor in andere, freie Seiten kopiert (2), damit keine Daten verloren gehen. Danach müssen die Verweise auf die gültigen Seiten angepasst werden, da diese nun an einer anderen Stelle gespeichert sind. Erst dann kann der Block gelöscht werden (3) und steht dem System wieder voll zur Verfügung. Der Prozess, der diese Säuberung vornimmt, wird Garbage Collector genannt.

[7] „Opferblock“, da er als „Opfer“ der Bereinigung/Säuberung ausgewählt wird

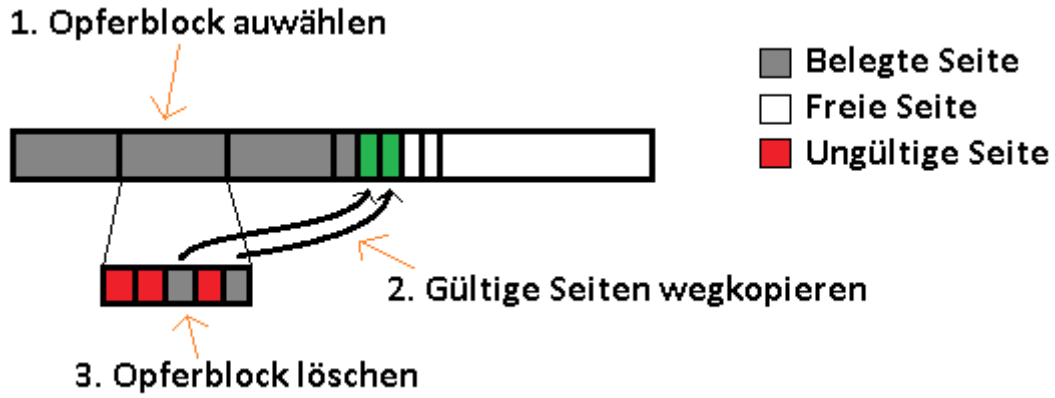


Abbildung 9: Garbage Collection: Grundsätzliche Funktionsweise

Die Vorgehensweise ist ähnlich wie bei Log-basierten Dateisystemen. In diesen werden die „Blöcke“ aus Flash (die Seiten zusammenfassen) jedoch als „Segmente“ (die Sektoren zusammenfassen) bezeichnet. Die Größe von Blöcken und Seiten von Flashspeichern ist von der Hardware vorgegeben, die Größe der Segmente der Log-basierten Dateisysteme können jedoch durch Software modifiziert werden. Es gibt auch einen weiteren Unterschied: während ungültige Segmente in den Log-basierten Dateisystemen direkt überschrieben werden könnten, müssen die Blöcke bei Flash-Speichern erst gelöscht werden, bevor dies möglich ist.

Es gibt mehrere Fragen, auf die die Garbage Collection eine Antwort finden sollte (vgl. [20], Abschnitt 2.2):

- Wann soll der Garbage Collector gestartet werden?
  Meistens wird dieser dann aktiviert, wenn die Anzahl an leeren Blöcken unter einen festgelegten Schwellenwert fällt oder spätestens dann, wenn für einen Schreibzugriff nicht genügend freier Platz vorhanden ist („on-demand“). Es ist jedoch auch möglich, ihn zusätzlich zu Leerlauf-Zeiten des Systems oder in festen Zeitabständen zu aktivieren. Das könnte allerdings wiederum für mobile Geräte problematisch sein, da unnötig zusätzliche Energie benötigt werden kann (vgl. [35], Seite 4).
- Wie viele Blöcke sollen gleichzeitig gesäubert werden?
  Je mehr Blöcke gleichzeitig gesäubert werden, desto mehr Daten können reorganisiert werden. Jedoch wird auch mehr Zeit zum säubern benötigt, wodurch

andere I/O-Zugriffe gestört werden können. Die meisten Implementierungen wählen daher nur einen Block.

- Welcher Block soll als victim block ausgewählt werden?
  Die Auswahl kann über mehrere Kriterien erfolgen, z.B. wie viele Datensätze innerhalb des Blocks bereits ungültig sind oder wann die Daten das letzte Mal geändert wurden. Dies wird als „victim selection algorithm“ bezeichnet (vgl. [20], Abschnitt 2.2 und [1], S. 268).
- Wohin sollen die noch gültigen Daten geschrieben werden?
  Es gibt unterschiedliche Möglichkeiten, wie die gültigen Daten neu geordnet werden können, um beispielsweise zukünftige Ausführungen der Garbage Collection zu optimieren, indem z.B. Daten mit gleicher Änderungshäufigkeit in die gleichen Blöcke geschrieben werden. Dies wird manchmal „data migration algorithm“ oder „data redistribution method“ genannt (vgl. [20], Abschnitt 2.2 und [1], S. 268).

4.1.1.1 Ziele

Folgende Ziele versucht ein Garbage-Collection-Algorithmus zu erreichen:

1. Möglichst geringe Behinderung anderer I/O-Operationen
   Es ist immer nur eine Operation gleichzeitig auf jedes Flash-Pack möglich. Wenn die Garbage Collection somit zu einem Zeitpunkt aktiv wird, in denen viele andere I/O-Operationen ausgeführt werden sollen, verzögert sie die anderen Zugriffe.
2. Anzahl der Lösch- und Schreibzugriffe möglichst gering halten
   Die Garbage Collection verschiebt Daten. Dadurch werden Blöcke gelöscht, auch wenn sie noch gültige Daten enthalten. Jeder Löschvorgang verringert die weitere Lebenserwartung des gelöschten Blocks.
3. Geringer Ressourcenverbrauch durch den Algorithmus
   Die Berechnung, welche Blöcke gesäubert werden sollen und wohin die noch gültigen Daten gesichert werden sollen, wird anhand von Algorithmen berechnet. Diese benötigen dafür Rechenzyklen der CPU oder des Firmware-Controllers, die zu dem Zeitpunkt anderen Prozessen nicht zur Verfügung stehen. Bei mobilen Geräten kann es dadurch zu einem erhöhten Stromverbrauch kommen.

Diese Ziele stehen im Konflikt miteinander. Um die Anzahl der Lösch- und Schreibzugriffe gering zu halten (2) wäre es sinnvoll, die Garbage Collection nur dann zu starten, wenn fast kein Speicherplatz mehr verfügbar ist. Dann ließen sich leicht Blöcke mit vielen ungültigen Seiten finden, womit nur wenige Daten kopiert werden müssen bevor ein Block gelöscht wird. Allerdings kann es dann passieren, dass zu einem Zeitpunkt viele neue Daten geschrieben werden sollen und dafür erst Platz geschaffen werden muss. Zu dem Zeitpunkt verzögert der Algorithmus dann diesen Schreibzugriff (1).

4.1.1.2 Einfluss des freien Speicherplatzes auf die Garbage Collection

Untersuchungen haben gezeigt, dass insbesondere der noch verfügbare Speicherplatz eines Flash-Speichers einen großen Einfluss auf seine Leistung hat (vgl. [32], [36], [37]): so nimmt die Schreibgeschwindigkeit aus Sicht der Anwendung bis zu 30% ab, wenn der Speicher statt zu 40%, zu 95% belegt ist (vgl. [38], Abschnitt 5.2). Der Grund liegt darin, dass bei einem fast vollen Speicher die Anzahl der Blöcke, die viele ungültige Daten beinhalten und durch den Garbage Collector schnell gesäubert werden können, sehr klein ist. Dadurch werden häufig ineffiziente Säuberungen durchgeführt, die zusätzliche Kopier- und Löschvorgänge auslösen.

4.1.1.3 Klassifizierung von Daten: Identifikation von *hot* und *cold data*

Garbage Collection kann effizienter durchgeführt werden, wenn die Daten klassifiziert werden und in entsprechend ihrer Schreibzugriffshäufigkeit in gleichen Blöcken gespeichert werden (vgl. [39], S. 23 und [40], Abschnitt I). Das liegt daran, dass sich häufig ändernde Daten im Flash nicht überschrieben, sondern in neue Blöcke geschrieben und damit die vorherigen Daten ungültig werden. Wenn viele Daten mit dieser Eigenschaft im gleichen Block gespeichert sind, werden nach einiger Zeit voraussichtlich nur noch wenige gültige Datensätze in dem Block vorhanden sein, wodurch die Garbage Collection nur wenige Daten aus dem Block kopieren muss, um wieder freien Platz zu schaffen. Daher ist es im Allgemeinen sinnvoll, die Daten in sich häufig ändernde Daten ("*hot data*") und solche, die selten geändert werden ("*cold data*") zu trennen. Allerdings ist es sehr aufwändig, genaue Zugriffsstatistiken über alle Daten in Flash-

blöcken zu erfassen und speichern. Daher wird in den Garbage-Collection-Algorithmen versucht, sich über verschiedene Mechanismen möglichst nah daran anzunähern, indem z.B. nur die Zugriffe über einen bestimmten Zeitraum erfasst werden.

Jeder Flash-Block kann nur bestimmt oft gelöscht werden, bevor er defekt wird. Wear-Leveling ist der Mechanismus, der dafür sorgen soll, dass sich die Anzahl der Löschvorgänge möglichst gleichmäßig über alle Blöcke verteilt. Daher spielt die Garbage Collection auch für das Wear-Leveling eine große Rolle, da der Garbage Collector die zu löschenden Blöcke auswählt und bestimmt, wohin die noch gültigen Daten eines zu löschenden Blocks kopiert werden sollen.

In den Unterkapiteln zu den verschiedenen Algorithmen wird später noch gezeigt, dass praktisch alle Garbage Collectoren diese Klassifikation der Daten in der einen oder anderen Weise durchführen. Die frühen Implementierungen haben dabei auf eher einfache Vorgehensweisen zur Identifikation von *hot* und *cold data* zurückgegriffen, in den meisten aktuellen wissenschaftlichen Artikeln zur Garbage Collection ist die Klassifizierung der wichtigste Bestandteil.

Es gibt auch Arbeiten, die sich losgelöst von Garbage Collection und Wear Leveling mit der *hot/cold data* Unterscheidung beschäftigen und Algorithmen vorschlagen. Die vorgeschlagenen Lösungen der Anfangszeiten hatten allerdings die Nachteile, dass sie entweder viel Hauptspeicher (vgl. [41], 1997) oder Rechenzeit benötigen (realisiert durch zwei LRU[8]-Listen, vgl. [42], 2002).

2006 wurde von Hsieh et al. (vgl. [39]) ein neuer Identifikations-Algorithmus mit mehrfachem Hashing vorgeschlagen, der diese Nachteile nicht hat und für die Realisierung in einem FTL vorgesehen ist. Prinzipiell spricht jedoch nichts dagegen, die Vorgehensweise auf Datei-Seiten von Flash-Dateisystemen zu übertragen. Hier werden die ankommenden Schreibbefehle auf LBA[9]-Blöcke gezählt. Diese LBAs werden parallel durch mehrere Hashfunktionen umgeformt, wie in Abbildung 10 dargestellt. Dort wird gezeigt, dass bei jedem Update eines LBA-Blocks die Adresse des Blocks durch eine

[8] LRU = Least recently used, also „am wenigsten kürzlich genutzt“. Es müssen hierfür die Zugriffe auf alle Elemente überwacht werden.

[9] LBA = Logical Block Address. Dieser Wert wird vom Betriebssystem zum Zugriff auf Block-Medien übergeben, bevor er intern in die physikalische, „echte“ Adresse übersetzt wird

Hashfunktion einer bestimmten Zeile in einer Hash-Tabelle zugeordnet wird und den dort vorhandenen Zähler um eins erhöht. Dies wird zusätzlich auch noch mit drei weiteren, anderen Hashfunktionen für die gleiche LBA durchgeführt. Der Grund dafür ist, dass es bei nur einer Hashfunktion zu Überschneidungen mit anderen LBAs kommen kann, deren Hash-Umformung zur gleichen Zeile in der Hash-Tabelle führt und sich die beiden LBAs diesen Eintrag in der Tabelle unabsichtlich teilen. Man spricht dann von einer Hashkollision. Dass jedoch alle Einträge zu einem LBA-Block (einer pro Hashfunktion, hier also vier) eine Überschneidung mit anderen LBA-Blöcken aufweisen, ist nahezu ausgeschlossen.

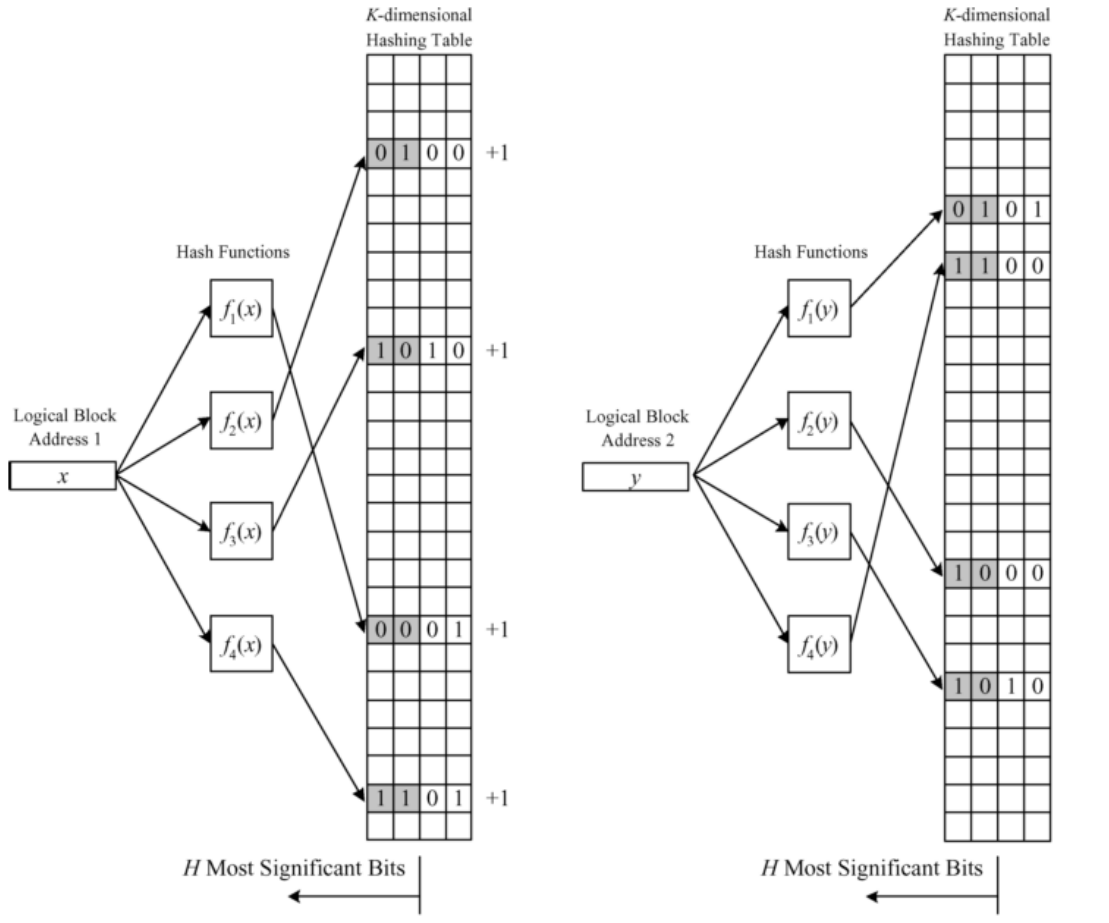


Abbildung 10: Vier Hashfunktionen speichern die Zugriffe auf LBAs in einer Hashtabelle (vgl. [39])

In konfigurierbaren, periodischen Abständen wird die Zahl der Schreibzugriffe im Zähler halbiert (durch eine bitweise Rechtsverschiebung in der Datenstruktur). Die ge-

nauen dazu nötigen Rechenvorgänge sind in der Veröffentlichung selbst dargestellt (vgl. [39]), hier sei nur erwähnt, dass die Zeit-Komplexität für die Operationen "Zählerstand für eine LBA lesen" oder "Zähler erhöhen" bei O(C) liegt, wobei C für die Anzahl der Bits des Zählers steht. Um alle Zähler gleichzeitig zu halbieren, liegt sie bei O(M|B) (M=Anzahl Zähler, B=Bit-Zahl eines Integers der zugrundeliegenden Hardware). Um Zählerüberläufe zu verhindern, wird vorgeschlagen, den Zähler beim Höchstwert entweder einzufrieren oder vor erneuter Erhöhung eine bitweise Rechtsverschiebung daran vorzunehmen.

Die Daten eines LBA-Blocks gelten in dieser Veröffentlichung dann als *hot*, wenn in allen zugehörigen Zählern die zwei höchstwertigen Bits 1 sind, hier also mindestens vier Zugriffe vor einer Zähler-Reduzierung stattgefunden haben. Mehrere Hashfunktionen werden deshalb benutzt, da Hashfunktionen prinzipiell einen großen Werteraum auf einen kleinen abbilden. Wird nur eine Hashfunktion benutzt, kann es daher leicht zu Hashkollisionen kommen, die zur fehlerhaften Einordnung einer Datenseite führen können, wenn unterschiedliche Adressen des großen Adressraums auf die gleiche Adresse im kleinen Adressraum abgebildet werden. Die Wahrscheinlichkeit für solche „*false positives*“[10] ist sehr deutlich reduziert, wenn mehrere Hashes genutzt werden. Ein LBA-Block gilt nur dann als *hot*, wenn alle Zähler zu diesem Block der Hash-Tabelleneinträge den Schwellenwert überschreiten.

Ein Leistungsvergleich ist schwierig, da das Ergebnis von sehr vielen Parametern abhängt, z.B. Art und Anzahl der genutzten Hash-Algorithmen, Bitgröße der Zähler, Größe der Hash-Tabelle oder dem Zeitabstand der Zähler-Halbierungen. Es wurde dann eine Einstellung auf einem Echtdaten-Trace simuliert, der wenig falsche Identifizierungen von *hot data* verursacht. Es wurde sowohl deutlich weniger Hauptspeicher als bei Chiang et al. (vgl. [41]) benötigt und auch nur ein Bruchteil der Rechenzeit des Algorithmus von Chang/Kuo (vgl. [42]). Weiterhin sollen weitere Optimierungen durch bessere Hash-Funktionen möglich sein, als die im Experiment eingesetzten.

Dieser Ansatz der *hot/cold*-Identifizierung wird allerdings, soweit bekannt, aktuell nirgends eingesetzt.

[10] „*false positive*“ bedeutet, dass etwas mit einer Eigenschaft identifiziert wird, die es tatsächlich nicht hat. In diesem Fall würde also ein Speicherblock als „*hot*“ erkannt, obwohl er eigentlich „*cold*“ ist.

Er bezieht nach Park/Du (vgl. [40], Abschnitt II c) allerdings durch die exponentielle Zählerreduzierung nicht ausreichend mit ein, wie kürzlich die Zugriffe stattgefunden haben, sondern nur deren Frequenz. Zur Verbesserung schlagen sie ihren eigenen Algorithmus vor, der mehrere Bloom Filter[11] zur Identifizierung von *hot data* einsetzt. Auch der Hashtabellen-Ansatz aus 2006 von Hsieh ist im Prinzip ein zählender Bloom Filter – jedoch eben nur einer. Durch den Einsatz mehrerer Bloom Filter mit jeweils kleinerer Tabellengröße braucht der gesamte Algorithmus noch weniger Hauptspeicher. So besteht die Hashtabelle jedes Bloom Filters bei Park/Du nur aus dem Schlüssel und einem einzigen Bit für den zugehörigen Wert, dafür werden z.B. 4 Hashtabellen gleichzeitig eingesetzt. Das kann man sich vorstellen wie in Abbildung 10 (a) dargestellt, nur dass es vier dieser Hashtabellen gibt, die jeweils nur genau eine ein Bit große Spalte besitzen.

Wird auf einen LBA geschrieben, werden in dem gerade aktiven Bloom Filter die entsprechend dazugehörigen Bits gesetzt. Folgt ein weiterer Schreibzugriff, werden im nächsten Bloom Filter die Bits gesetzt (Round Robin). Sollten alle Bits in allen Filtern bereits gesetzt sein, gelten die Daten automatisch als *hot*. Nach einer festgelegten Periode, hier nicht in Zeit gemessen sondern in der Menge der Schreibzugriffen seit dem letzten Wechsel, wird ebenfalls im Round Robin Verfahren einer der Filtertabellen komplett zurückgesetzt.

Dadurch ist zusätzlich zur Frequenz des Zugriffs auf LBAs feststellbar, wie kürzlich ein Zugriff erfolgte: sind die entsprechende Bits in der aktuell aktiven Tabelle gesetzt, ist der Zugriff in dieser Zeitperiode geschehen. Daher wird allen Bloom Filtern eine Gewichtung zugewiesen. Der Filter mit der zuletzt zurückgesetzten Tabelle erhält das höchste Gewicht, der zuvor gelöscht ein etwas kleineres, und so weiter. Dadurch wird dann der *hot data index* erstellt, der jedem in den Tabellen erfassten LBAs einen Wert zuweist, aus dem sich ergibt, ob er als *hot* gilt. Der Wert setzt sich sowohl aus der Frequenz (durch die gesetzten Bits) und der Kürzlichkeit (durch die Gewichtung der Tabelle, in denen die Bits gesetzt sind) der Schreibzugriffe zusammen.

Der Vorteil der Bloom Filter von Park/Du gegenüber der Hashtabelle von Hsieh et al. liegt darin, dass die zeitliche Lokalität von Zugriffen besser erfasst werden kann

[11] „Bloom Filter“ wurden 1970 von Burton Bloom erfunden und nach ihm benannt. Es ist der Vorgang, ein Datum über mehrere Hashfunktionen in einen kleineren Adressraum abzubilden. Die Funktionsweise ist analog zu Abbildung 10

und eine qualitativ noch bessere Identifikation von *hot data* möglich ist. Es werden durch die Vorgehensweise auch deutlich weniger Daten fälschlich als *hot* identifiziert (vgl. [40], Fig. 13).

Im Ergebnis zeigte sich, dass der Algorithmus von Park/Du nur circa die Hälfte an Hauptspeicher benötigt und bis zu 65% schneller berechnet werden kann als im Hash-tabellen-Ansatz. Auch dieses Identifikations-Schema wird bislang, soweit bekannt, in keinem der Flash-Dateisysteme eingesetzt.

4.1.2 Garbage Collection Algorithmen

4.1.2.1 Greedy (1992)

Der älteste bekannte Garbage Collection Algorithmus wird „Greedy“[12] genannt. Bei ihm wird immer der Block zur Säuberung gewählt, der die meisten ungültigen Seiten enthält (vgl. [43], S. 91). Die Informationen, welche Seiten innerhalb der Blöcke ungültig sind, wird im Hauptspeicher vorgehalten und jedes Mal aktualisiert, wenn ein bestehende Seite überschrieben oder gelöscht wird (vgl. [32], S. 40). Die noch gültigen Seiten werden in der gleichen Reihenfolge in neue Blöcke geschrieben, in der sie gelesen wurden. Diese Vorgehensweise wurde für Log-basierten Dateisystemen erfunden (vgl. [32], S. 33), dort wurden jedoch mehrere Blöcke gleichzeitig gesäubert und dabei die einzelnen Sektoren nach Zugriffshäufigkeit getrennt. Dies wird für Flashspeicher nicht getan, da in den 1990er-Jahren die Flashmodule noch in nur wenige, große Blöcke unterteilt waren und dieses Vorgehen daher zu lange dauern würde. Jedoch hat der freie Speicherplatz einen starken Einfluss auf den Overhead, den der Greedy-Algorithmus erzeugt: ab ca. 80% belegtem Speicherplatz werden für jeden einzelnen Schreibvorgang ca. 3-10 zusätzliche Schreibvorgänge zur Säuberung von Segmenten[13] benötigt. Der Grund dafür ist, dass mit zunehmender Speicherbelegung immer mehr gültige Seiten in den zu säubernden Blöcke sind, die kopiert werden müssen, bevor der Block gelöscht werden kann. Dies wurde auch theoretisch hergeleitet und durch Testreihen bestätigt (vgl. [37] , S. 13 und S. 19/Figure 8).

[12] engl. greedy = dt. gierig

[13] Hier wird der Begriff „Segment“ verwendet, da dieser bei der Erfindung des Algorithmus für Log-basierte Dateisysteme genutzt wurde. Für Algorithmen, die auf Flash eingesetzt werden, kann bedenkenlos „Block“ synonym eingesetzt werden.

Deshalb lässt es z.B. das Flash-Dateisystem eNVy nicht zu, dass mehr als 80% der Speicherkapazität belegt werden, wenn Greedy für die Garbage Collection gewählt wird (vgl. [43], S. 90)

Der Algorithmus ist auch im Flash File System implementiert (vgl. [44]).

4.1.2.2 Cost-Benefit (1992)

Wegen des großen I/O-Overheads des Greedy-Algorithmus bei einer hohen Speicherplatzbelegung, wurde für das LFS-Dateisystem noch ein anderer Garbage Collection Algorithmus namens „Cost-Benefit“[14] diskutiert (vgl. [32], S. 38). Dieser wählt den zu säubernden Block[15] durch die folgende Formel aus:

$$\frac{a \cdot (1-u)}{1+u} \quad \text{oder umgangssprachlich} \quad \frac{Zeitfaktor \cdot (Nutzen\,der\,Blocksäuberung)}{Kosten\,der\,Blocksäuberung}$$

Die Blöcke mit dem höchsten Ergebnis der Formel werden zur Säuberung ausgewählt. Das a steht für einen Zeitfaktor, genauer gesagt die Zeit seit der letzten Änderung irgendeines Datensatzes innerhalb des Blocks. Das u ist der Füllgrad, also das Verhältnis von gültigen Datensätzen zur möglichen Gesamtdatensatzzahl innerhalb des Blocks. Vereinfacht gesagt bedeutet das, dass dieser Algorithmus bei Blöcken, die vor kurzem bereits Änderungen geschrieben haben, etwas länger mit der Säuberung wartet, da in naher Zukunft mit hoher Wahrscheinlichkeit weitere Datensätze ungültig werden.- Denn ca. 90% der Schreibvorgänge finden auf kürzlich veränderte Dateien statt (vgl. [32], S. 40).

Diese andere Vorgehensweise wirkt sich ab ca. 40% belegtem Speicherplatz auf die Leistungsfähigkeit gegenüber Greedy aus (vgl. [32], Fig. 7 auf S. 40). Das liegt daran, dass Greedy häufig Blöcke säubert, bei denen in Kürze weitere Seiten ungültig geworden wären und den Säuberungsaufwand vermindert hätten. Der Cost-Benefit-Algorithmus bezieht das durch den Zeitfaktor mit ein.

Diese Vorgehensweise hat auch Nachteile. Nach vielen Läufen der Garbage Collection vermischen sich *hot* und *cold data* in den Blöcken, da die noch gültigen Seiten beim Kopieren nicht getrennt werden (vgl. [20], S. 107). Die *cold data* muss bei zu-

[14] engl. cost-benefit = dt. Kosten-Nutzen

[15] Auch bei Cost-Benefit: der ursprüngliche Begriff „Segment“ ist historisch bedingt und wird bei Flash als „Block“ bezeichnet

künftigen Garbage-Collection-Vorgängen dann immer mit kopiert werden und erzeugt somit unnötige I/O-Vorgänge.

Dieser Garbage-Collection-Algorithmus wird im Sprite-LFS-Dateisystem, im Flash File System und im F2FS eingesetzt (vgl. [32], [44] und Kap. 5.5).[16]

4.1.2.3 Dynamic Data Clustering (DAC, 1999)

DAC (Dynamic dAta Clustering) ist eine Art logische Zwischenschicht, die „Flash Memory Server“ genannt wurde (vgl. [1], S. 272). Durch sie wird der Flashspeicher in mehrere Regionen aufgeteilt. Innerhalb dieser Regionen ist dann ein Garbage-Collection-Algorithmus nur für seine Region aktiv.

Ziel des Ansatzes war es, Flashblöcke möglichst selten löschen zu müssen, da dies sehr zeitaufwändig ist. Das soll erreicht werden, indem Daten nach ihrer Schreibzugriffshäufigkeit getrennt werden um diese möglichst zusammen in den gleichen Blöcken zu speichern. Dadurch werden häufig geänderte Daten in solchen Blöcken schnell ungültig und die Garbage Collection muss deswegen beim Säubern weniger gültige Daten in andere Blöcke kopieren und kann einfacher freien Platz bereitstellen, als es bei vermischten Daten möglich wäre. Die Anzahl der Flash-Block-Lösch- und Kopiervorgänge ist dadurch geringer. Da NAND-Flashspeicher damals noch keine Rolle gespielt hat, bezieht sich die Vorgehensweise auf NOR-Flash, der bitweise Programmierung innerhalb jedes Flash-Blocks zulässt und somit nicht in einzelne Flash-Seiten unterteilt ist. Der Begriff „Daten“ bezieht sich daher auf die kleinste vom Dateisystem geschriebene Einheit, ein Datenblock.

Der Flashspeicher wird in mehrere Regionen unterteilt. Daten, die ähnlich oft geändert werden, sollen dabei in der gleichen Region gespeichert werden. Dies kann man sich wie in Abbildung 11 gezeigt vorstellen.

---

[16] Im Flash File System ist die zu maximierende Formel leicht verändert: $\frac{a \cdot (1-u)}{2 \cdot u}$ Das wird damit begründet, dass der Nenner die Kopier-Kosten der Block-Säuberung darstellt und diese sich von einem Flashspeicher zu einem Dateisystem auf einer Festplatte unterscheidet

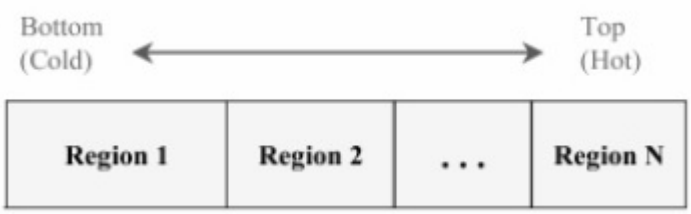


Abbildung 11: Aufteilung des Flash-Speichers in Regionen (vgl. [1], Fig. 1)

Neue Daten starten immer in der kleinsten Kategorie. Die Klassifikation der Daten kann durch zwei Ereignisse verändert werden: bei der Garbage Collection und beim Ändern der Daten. Werden Daten geändert, dann werden diese Daten um eine Region hochgestuft (z.B. von Region 2 in Region 3). Wenn bei der Säuberung eines Flashblocks noch gültige Daten kopiert werden, dann werden diese eine Region abgestuft, da sie sich offenbar schon eine gewisse Zeit lang nicht mehr verändert haben. Somit werden sich mit der Zeit selten ändernde Daten in der Region 1 ansammeln und häufig geänderte Daten in der höchsten Region.

Um festzustellen, in wie viele Regionen ein Flashspeicher eingeteilt werden sollte, wurden Simulationen mit verschiedenen Parametern durchgeführt, z.B. Flashspeichergröße, Anzahl der Regionen, Speicherplatzbelegung, etc. (vgl. [1], S. 278ff.). Dazu mussten die Säuberungskosten definiert werden, die Chiang et al. hergeleitet haben und dann zu folgendem Ergebnis bekommen sind (Details dazu in [1], S. 275f.)[17]:

$$\text{Vereinfachte Säuberungskosten} = \text{AnzahlLöschen} + \left(\frac{\text{Gesamtzahl\_Kopiert\_Datenblöcke}}{\text{Anzahl\_Datenblöcke\_pro\_Flashblock}} * \text{SchreibenZuLöschenVerhältnis} \right)$$

Simulationen haben ergeben, dass ab dem Einsatz von zwei Regionen, also der erstmaligen Trennung von selten und häufig veränderten Daten, eine starke Verringerung der Garbage-Collection-Säuberungskosten eintritt, vgl. Abbildung 12.[18] Das liegt insbesondere daran, dass Flashblöcke seltener gelöscht und noch gültige Datensätze innerhalb eines Blocks seltener kopiert werden müssen. Die Ursache dafür ist, dass es durch diese Trennung von *hot* und *cold data* bei der Säuberung von Blöcken nicht dazu kommt, dass häufig auch unveränderte Daten jedes mal zusammen mit der *hot*

[17] In der Veröffentlichung werden die Begriffe leicht anders als sonst im Flashumfeld verwendet. „Segment“ bezeichnet dort einen Flashblock und mit „Block“ ist eine von Software festgelegte Blockgröße gemeint (vgl. [1], Fußnote auf S. 267)

[18] Der dritte Garbage-Collection-Algorithmus, CAT, wird in dieser Arbeit nicht beschrieben

data wieder in den gleichen Block geschrieben. Sobald mehr als drei Regionen eingesetzt wurden, tritt keine wahrnehmbare, weitere Verbesserung mehr ein. Da jede Region selbst freie Blöcke für den in ihr arbeitenden Garbage Collector vorhalten muss, ist der noch freie Speicherplatz fragmentierter. Diese Fragmentierung führte zu zusätzlichen Block-Löschvorgängen und verschlechtere das Ergebnis ab ca. sieben eingesetzten Regionen wieder (vgl. [1], S. 279).

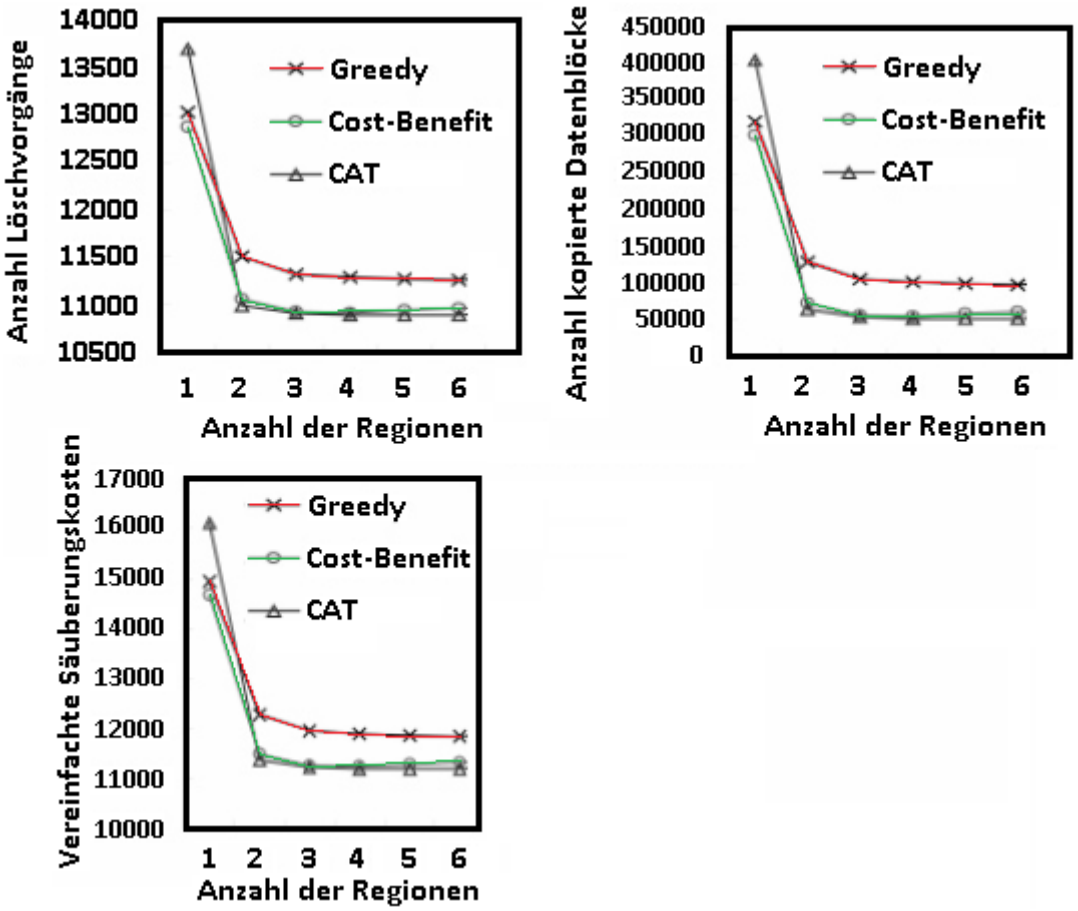


Abbildung 12: DAC-Performancevergleich (basierend auf [1], S.278-279)

Die Betrachtung lässt jedoch außer Acht, dass die DAC-Zwischenschicht selbst auch Systemressourcen benötigt. So werden im RAM für jede Region 12 Bytes, für jeden Flashblock 17 Bytes und für jeden Datenblock 13 Bytes benötigt. Bei der damaligen Größe von Flashspeicher im Bereich von wenigen Megabyte wurde dafür weniger als 100 KByte belegt. In einem aktuellen embedded System (z.B. in einem Smartpho-

ne) mit 16 GB Speicherkapazität, 128 KByte pro Block und 2 KByte pro Datenblock würde DAC aber über 55 MB RAM belegen.

Aus diesen Ergebnissen kann abgeleitet werden, dass eine Unterteilung der Daten in *hot* und *cold* starke Performance-Verbesserungen ermöglicht, eine noch feinere Abstufung jedoch nicht unbedingt notwendig ist.

4.1.2.4 Cost-Age-Time with Age-sort (CATA, 2006) und Cost-Benefit with Age-Sort (CBA, 2006)

CATA (vgl. [20]) und CBA (vgl. [45]) sollen das Problem lösen, dass bei einem Zugriffsverhalten mit hohen Lokalitätseigenschaften nach kurzer Zeit Seiten mit sich wenig verändernden Daten („*cold data*“) und solche mit häufigen Änderungen („*hot data*“) sich vermischen. Dann würde die *cold data* bei der Säuberung durch den Garbage Collector häufig kopiert, da sie in den selben Blöcke wie die *hot data* nach der vorherigen Verschiebung gespeichert wurde. „Beide“ Algorithmen sind 2006 von den gleichen Hauptautoren vorgestellt worden und unterscheiden sich praktisch nicht, außer durch den Namen und leicht geändertem Wortlaut in den veröffentlichten Artikeln. Alle folgenden Aussagen beziehen sich daher auf beide Algorithmen, es sei denn es wird explizit auf einen Unterschied hingewiesen.

Um zu verhindern, dass sich *hot* und *cold data* vermischen, sollte der Garbage Collector mehrere Blöcke gleichzeitig zum Löschen wählen und beim Verschieben der noch gültigen Daten diese nach *hot* und *cold* unterscheiden und in unterschiedlichen Blöcken speichern. Da das Löschen von Blöcken lange dauert, könnte das zur Verlangsamung „echter“ I/O-Zugriffe führen, wenn mehrere auf einmal gelöscht würden. Dieses Problem soll durch eine Vorhersage-Routine in CATA/CBA gelöst werden. Die funktioniert, wie in Abb. 13 gezeigt.

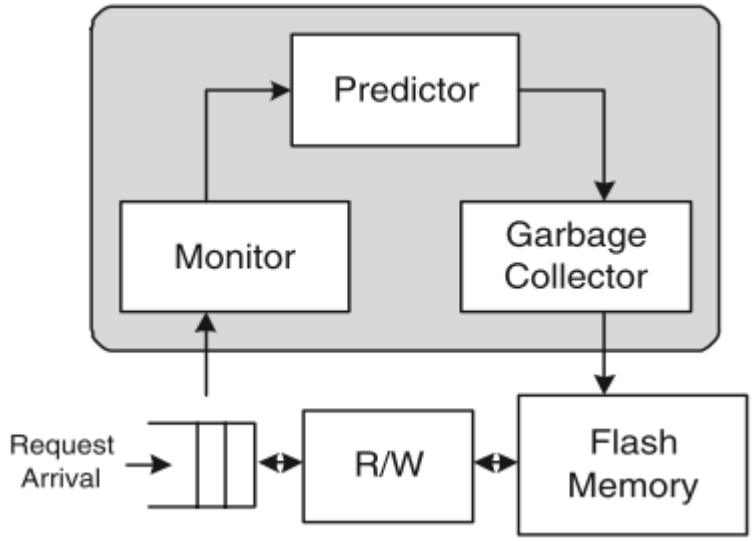


Abbildung 13: CATA Garbage Collection Architektur (vgl. [20]*, Figure 3)*

Der Monitor zählt die eingehenden I/O-Zugriffe. Die Vorhersage-Routine („Predictor“) schätzt daraus die Anzahl der erwarteten I/O-Zugriffe der unmittelbaren Zukunft. Die funktioniert prinzipiell, indem die Anzahl der ankommenden I/O-Zugriffe in einem festen Intervall gemessen und über einen AR(1)-Modell[19] korreliert werden. Es geht der Wert des vorhergehenden Intervalls und ein Rauschterm in das Ergebnis der Vorhersage-Routine mit ein. Details mit allen mathematischen Formeln dazu sind in der Veröffentlichung (vgl. [20], S. 7) detailliert dargestellt.

Die Anzahl der zu säubernden Blöcke wird wie folgt festgelegt: sagt diese Routine bei Start der Garbage Collection voraus, dass viele Zugriffe erwartet werden, wird nur einer oder gar kein Opferblock ausgewählt. Andernfalls werden zwei bis drei Blocks gesäubert.

Es werden die Blöcke gewählt, die in der folgenden Formel den höchsten Wert erreichen (nur bei CATA; im CBA-Artikel wird dazu keine Aussage getroffen):

$$\frac{1-u}{1+u} \cdot a \cdot \frac{1}{erasecount}$$

Das *u* steht für das Verhältnis von gültigen zur Gesamtzahl möglicher Seiten in einem Block. Das Alter („*a*“) gibt die Zeit an, die seit der letzten Änderung des Blocks vergangen ist und der *erasecount* steht für die Anzahl, wie oft der Block bereits gelöscht wurde. Dieser erasecount sorgt in der Formel dafür, dass häufig gelöschte Blö-

[19] AR steht für „autoregressive model“, ein Modell aus der Statistik und Messtechnik. Es dient für Prozesse mit zufälligen Einflüssen, die über einen zufälligen Rauschwert simuliert werden.

cke seltener gesäubert werden, und somit durch CATA auch *Wear Leveling* (s. nächstes Unterkapitel) durchgeführt wird. Sobald alle Blöcke bestimmt sind, werden die noch gültigen Seiten, die kopiert werden müssen, nach der Zeit der letzten Modifikation sortiert. Die am längsten unveränderten Seiten werden dabei zuerst geschrieben. Je mehr Blöcke gleichzeitig gesäubert werden, desto eher sorgt diese Vorgehen dafür, dass Seiten mit ähnlichem Änderungsverhalten in den gleichen Zielblöcken landen.

Die Tests wurden auf einem zu 90% mit Daten gefüllten Flashspeicher durchgeführt, da die Garbage Collection nur bei sehr vollem Speicherplatz einen wirklichen Performanceunterschied zeigt. Die Leistungsfähigkeit von CATA/CBA wurde dabei mit dem Greedy- und dem Cost-Benefit-Algorithmus verglichen. Dabei wurde festgestellt, dass bei einer hohen Lokalität der Zugriffe (ab ca. 80% lokal, 20% zufällig) der CATA/CBA-Algorithmus bessere Ergebnisse liefert, wenn drei oder mehr Blöcke gleichzeitig gesäubert werden, d.h. wenn wenige Zugriffe in der unmittelbaren Zukunft vorhergesagt werden, dann ist der I/O-Durchsatz des Flashspeichers höher. Bei weniger gleichzeitig gesäuberten Blöcken liegt CATA/CBA mit den anderen ungefähr gleichauf. Hier zeigt sich, dass die Trennung der Seiten nach ihrem Änderungsverhalten erst ab drei Blöcken einen merkbaren Effekt auf das Ergebnis hat.

4.1.2.5 Harmonia (2011)

Dieser Garbage-Collection-Algorithmus ist für ein spezielles Einsatzgebiet gedacht: er soll die Performance in SSD-RAID[20]-Arrays verbessern (vgl. [46], [47]). Eine SSD ist ein Speichermedium für Computer, welche intern Flashspeicher zur Datenhaltung verwendet und als Ersatz für die bislang üblichen Festplatten (insbesondere in PCs und Notebooks) gedacht ist.

Die Gesamtgeschwindigkeit eines RAIDs hängt von der langsamsten Platte im Verbund ab. Daher verlangsamt eine SSD, auf der gerade Garbage Collection durchgeführt wird, den gesamten RAID, auch wenn die anderen SSDs nichts zu tun haben und theoretisch Daten liefern/speichern könnten. Die Garbage Collection selbst wird von der Firmware der einzelnen SSDs gesteuert. Um diesem Problem entgegenzuwirken, schlagen die Autoren vor, die Garbage Collection über alle SSDs im RAID zu synchronisieren. Um das umsetzen zu können, muss der RAID-Controller verändert werden,

[20] SSD steht für „Solid State Drive“; RAID steht für „Redundant Array of Inexpensive/Independent Disks“ und bedeutet, dass mehrere Speicherplatten zusammengeschaltet werden

so dass er Befehle zur Garbage Collection an die angeschlossenen SSDs schicken kann. Die SSDs müssen auch verändert werden. Sie benötigen zwei neue Anweisungen: einerseits müssen sie über ihren aktuellen Fragmentierungs-Status auf Anforderung Auskunft erteilen und andererseits den RAID-Controller informieren, wenn Garbage Collection notwendig ist.

Der Harmonia-Algorithmus bietet zwei mögliche Vorgehensweisen. In der Einstellung „reaktiv“ reagiert er auf die Information von SSDs, dass sie Garbage Collection benötigen. Dann kann der RAID-Controller alle SSDs anweisen, einen Garbage-Collection-Zyklus zu starten oder alternativ noch etwas warten, bis weitere SSDs ihn benachrichtigen. Bei „proaktiv“ erhält der RAID-Controller in regelmäßigen Abständen von den SSDs Informationen über die interne Fragmentierung, die Anzahl freier Blöcke und ECC-Fehler bei Lesevorgängen. Durch diese Informationen kann der RAID-Controller zu geeigneten Zeiten wieder parallel den Garbage-Collection-Algorithmus auf allen SSDs starten.

In Tests mit einem von Microsoft angepassten SSD-Simulator (vgl. [48], Kap. 4.1) konnte gezeigt werden, dass durch solche Anpassungen eine bis zu 69% schnellere Reaktionszeit gegenüber SSD-RAIDs ohne Harmonia möglich war. Insbesondere bei kurzzeitig auftretenden Schreibvorgängen mit vielen Daten („burst“) zeigte sich diese starke Performanceverbesserung (vgl. [46], Abschnitt VI).

4.2 Wear Leveling

Wie bereits in den technischen Grundlagen beschrieben, können einzelne Flash-Zellen nur begrenzt oft gelöscht und neu programmiert werden, bevor sie defekt werden. Die Hersteller gehen aktuell von ca. 10.000 bis 1.000.000 Zyklen aus (vgl. Tabelle 1 auf S. 9).

Deswegen ist es wünschenswert, einen Mechanismus zu implementieren, der dafür sorgt, dass die Anzahl der Löschvorgänge auf jeden Block möglichst gleich ist („Abnutzung“). Dieser Vorgang wird „Wear Leveling“[21] genannt.

Früher wurde Flashspeicher hauptsächlich für einige mobile Anwendungsbereiche eingesetzt, z.B. für Digitalfotografie oder Mobiltelefone. Diese Geräte haben den Speicherplatz meist sehr gleichmäßig genutzt und somit auch abgenutzt. Daher spielte

[21] Engl. „wear“ = Abnutzung, Haltbarkeit; Engl. „Leveling“ = Einebnen, Gleichmachen; Frei übersetzt kann Wear-Leveling somit als „Angleichen der Abnutzung“ bezeichnet werden

Wear Leveling zu der Zeit nur eine untergeordnete Rolle. Seit jedoch Flashspeicher in großen Stückzahlen auch für Universalcomputer mit ganz anderem Zugriffsverhalten eingesetzt wird (z.B. SSD-Massenspeicher für Notebook- und Desktop-Rechner), ist ein Algorithmus zur gleichmäßigen Abnutzung der Flash-Zellen erforderlich.

Der Algorithmus hat mehrere Ziele:

1. es soll dafür gesorgt werden, dass die Flash-Blöcke möglichst gleich oft gelöscht werden,
2. bei der Verteilung der Flash-Block-Abnutzung dürfen nur möglichst wenige Schreib- und Löschvorgänge durchgeführt werden, um nicht zum Teil des Wear-Problems zu werden und
3. es soll möglichst wenig Overhead erzeugt und Ressourcen belegt werden

Weiterhin werden Flash-Blöcke in *young* („jung“) und *old* („alt“) unterteilt. Jung bedeutet, dass dieser Block noch nicht oft gelöscht wurde und somit noch oft wiederverwendet werden kann. Alt bedeutet entsprechend im Gegenteil, dass der Block schon häufig neu gelöscht wurde.

Dynamisches und statisches Wear Leveling

Im dynamischen Wear Leveling werden nur die Blöcke mit einbezogen, auf die demnächst geschrieben werden soll (vgl. [49], Kap. 2.5). Beispielsweise geschieht das dann, wenn ein neuer freier Block vom Flash-Dateisystem oder der FTL angefordert wird. Es wird dann der Block aus der Menge aller freien Blöcke verwendet, der bislang noch die wenigsten Löschzyklen durchlaufen hat.

Im Gegensatz dazu umfasst statisches Wear Leveling alle Blöcke, d.h. auch die, deren Inhalt sich schon lange nicht mehr geändert hat und die daher nie gelöscht wurden, um wieder als freier Block verfügbar zu sein.

Letzteres ist wichtig, da sonst die Blöcke von Daten, die sich sehr selten ändern (z.B. die Dateien des Betriebssystems), kaum abgenutzt werden, während alle übrigen Blöcke weit übermäßig belastet werden. Weiterhin gibt es durch die immer weiter verkleinerten Zellen einen Effekt, der dazu führen kann, dass auch bei Lesevorgängen der Inhalt benachbarter Zellen nach einiger Zeit beeinflusst werden kann (vgl. Kap. 2.4.3).

Die Daten gelegentlich neu zu schreiben beugt dem Problem vor. Daher ist statisches Wear Leveling auch aus diesem Grund wichtig.

4.2.1 Wear-Leveling Algorithmen

Über die Jahre wurden mehrere Wear-Leveling-Algorithmen vorgeschlagen, von denen repräsentativ einige vorgestellt werden sollen.

4.2.1.1 Hot-Cold-Swap (1994)

Die einfachste Möglichkeit, für eine gleichmäßige Abnutzung der Flash-Blöcke zu sorgen, ist die Daten zwischen dem ältesten und jüngsten Block zu vertauschen, sobald die Differenz der Anzahl der Löschzyklen zwischen ihnen einen gewissen Schwellenwert überschreitet.

Daran wurde kritisiert, dass durch die vielen Kopiervorgänge beim Vertauschen der Blöcke zu viele I/O-Operationen stattfinden und die Leistungsfähigkeit des Flashspeicher dadurch zu stark beeinflussen (vgl. [50], Kap. 4.2.4). Dem kann entgegengehalten werden, dass dies nur dann ins Gewicht fällt, wenn der Schwellenwert zu gering gewählt ist. Da jeder Flashblock zwischen mindestens 10.000 mal gelöscht werden kann, spricht nichts dagegen, den Schwellenwert auf 1.000 und mehr festzulegen. Dann wird zwar immer noch der I/O-Overhead beim Tauschen entstehen, jedoch wird es nicht weiter auffallen, wenn der gesamte I/O betrachtet wird, da es nur noch ein seltener Vorgang ist.

So ist auch eines der ältesten Flash-Dateisysteme, eNVy, vorgegangen und hat diesen Mechanismus mit dem Schwellenwert von 100 implementiert (vgl. [43], S. 92). Auch das aktuelle Flash-Dateisystem UBIFS nutzt diese Vorgehensweise mit einem Schwellenwert von 5000 (vgl. Kap. 5.4).

4.2.1.2 Turn-Based Selection (2001)

Dieser Algorithmus ist eigentlich ein Garbage Collector, der zusätzlich Wear Leveling durchführen soll. Es gibt für den Algorithmus zwei relevante Listen der vorhandenen Flashblöcke, dirty und clean. In der „clean“-Liste befinden sich alle Blöcke, die

nur gültige Daten beinhalten. Ist auch nur ein Datum ungültig, wird der Block stattdessen der „dirty“-Liste zugeordnet. Wenn der Garbage Collector aktiviert wird, wählt dieser einen zu säubernden Block per Wahrscheinlichkeit aus einer der beiden Listen aus, z.B. 90-10. Dies würde bedeuten, dass zu 90% ein Block aus der „dirty“-Liste und zu 10% ein Block aus der „clean“-Liste gewählt wird. Dann wird die Garbage Collection durchgeführt, d.h. noch gültige Datensätze kopiert und der Block gelöscht.

Der Sinn in dieser auf Wahrscheinlichkeit basierenden Auswahl ist, dass auch statisches Wear Leveling durchgeführt wird und somit Blöcke mit einbezogen werden, deren Inhalt sich nicht ändert. Weiterhin muss keine Statistik darüber geführt werden, wie oft ein Block bereits gelöscht wurde.

An dem Ansatz wird kritisiert, dass ein Block, der aus der „clean“-Liste gewählt wird, nicht unbedingt ein junger Block sein muss und die Abnutzung der Blöcke zu ungleichmäßig geschieht (vgl. [23], Kap. 3.2).

Diese Art von Wear Leveling wird im JFFS2-Dateisystem eingesetzt (vgl. Kap. 5.2).

4.2.1.3 Dual Pool (2007)

2007 schlug Chang einen „Dual Pool“ genannten Wear-Leveling-Algorithmus vor (vgl. [23]). Die grundlegende Idee dahinter ist, dass *cold data*, die sich selten ändert, bevorzugt in alte Blöcke migriert werden sollte, während *hot data* besser in jungen Blöcken aufgehoben ist.

Dies wird durch zwei „Pools“ von Flash-Blöcken realisiert, einen „hot pool“ und einen „cold pool“. *Hot data* wird dabei immer in den *hot pool* geschrieben und *cold data* in den *cold pool*. Die Blöcke jedes Pools werden in einer Liste geführt, die nach der Anzahl der Löschzyklen jedes Blocks sortiert sind. Bei jedem Schreibvorgang wird überprüft, ob der Block mit der höchsten Löschanzahl des *hot pools* um mehr als ein zu definierender Schwellenwert des am seltensten gelöschten Blocks des *cold pools* abweicht. Wenn das der Fall ist, wird der Inhalt der beiden verglichenen Blöcke ausgetauscht und jeder Block dem jeweils anderen Pool zugeordnet, d.h. der Block, der vorher dem *hot pool* angehörte, ist nun im *cold pool* und umgekehrt.

Durch dieses Vorgehen ist der häufig gelöschte Block mit mutmaßlicher *cold data* beschrieben und dem *cold pool* zugeordnet, wird demnach in naher Zukunft wegen seines hohen Löschzählers dort bleiben und daher nicht mehr so schnell abgenutzt.

In der Veröffentlichung wurde mit dem Schwellenwert 8 getestet. Im Vergleich mit 8 anderen Wear Leveling Algorithmen zur damaligen Zeit erreichte er die gleichmäßigste Abnutzung und verbrauchte dabei vergleichsweise wenige CPU- und Hauptspeicher-Ressourcen (vgl. [23], Table 3).

Allerdings spricht meines Erachtens nichts dagegen, einen Schwellenwert von weit über 500 zu wählen. Dadurch ist die Abnutzung nicht mehr ganz gleichmäßig, allerdings werden dann weit weniger Kopier- und Löschvorgänge durch das Wear Leveling selbst verursacht. Bei durchschnittlich über 55.000 Löschzyklen, die ein Block übersteht, ist eine Abweichung von 500 nicht bedeutend.

4.2.1.4 Lazy Wear Leveling (2012)

Es zielt auf eine Implementierung in SSD-Firmware-Controllern ab, wodurch die verfügbaren Rechenzeit- und Hauptspeicher-Ressourcen für die Flash-Management-Mechanismen stark eingeschränkt sind (vgl. [13], Abschnitt 1). Die Autoren der Veröffentlichung stellen fest, dass die anderen Wear Leveling Algorithmen ihre eigenen Datenstrukturen benötigen, die im Hauptspeicher gehalten werden müssen. Sie wollen diesen Overhead einsparen, indem sie verfügbare Flash-Verwaltungsinformationen aus dem Adressübersetzungsmechanismus der Firmware mit benutzen.

Die Vorgehensweise von Lazy Wear Leveling (Lazy) basiert einigen Feststellungen. Zum einen ist das Schreibverhalten von Datenspeichern für PCs üblicherweise durch starke Lokalität geprägt. Die meisten Zugriffe erfolgten auf eine kleine Anzahl von logischen Sektoren des Dateisystems (vgl. [13], Figure 3 und [51], S. 863). Zum anderen beinhalten solche Blöcke im Flashspeicher, die bei einem hybridem Adressübersetzungsverfahren (vgl. Kap. 4.4.2) über keine zugeordneten Log Blöcke verfügen, sehr wahrscheinlich *cold data*.

Diese beiden Annahmen nutzt Lazy, um Daten mit einem bestimmten, erwarteten, zukünftigen Zugriffsverhalten in die Blöcke zu schreiben, die davon jeweils in Hinblick auf das Wear Leveling am meisten profitieren. Ein Beispiel: ist ein Flashblock bereits sehr oft gelöscht worden, verglichen mit der durchschnittlichen Löschanzahl aller anderen Blöcke, wird Lazy *cold data* aus einem länger nicht mehr veränderten Block in ihn kopieren.

Im Detail geht der Algorithmus dabei wie folgt vor (vgl. [13], Algorithm 1). Jedes mal, wenn der Garbage Collector der FTL einen victim block (vgl. Kap. 4.1.1) löschen will, wird der Lazy-Algorithmus aufgerufen.

Dieser prüft, ob der Löschzähler des zu löschende Blocks vergleichen mit der durchschnittlichen Löschanzahl aller anderen Blöcke über einem Schwellenwert liegt. Wenn nicht, beendet sich der Lazy-Algorithmus an dieser Stelle und greift nicht ein.

Sollte der Schwellenwert überschritten sein, iteriert Lazy so lange über logische Blocknummern der Blöcke, bis es einen Block findet, der keinen zugewiesenen Log Block hat. Dazu greift er auf die Daten der Adressübersetzung der FTL im Controller zu. Sobald ein solcher Block gefunden ist, wird der victim block gelöscht und der Inhalt des soeben gefundenen Blocks in ihn kopiert sowie die entsprechenden Block-Mappings angepasst. Lazy gibt dann den Block, dessen Daten gerade kopiert wurden und den Inhalt dieses Blocks somit ungültig machen, als neuen victim block an den Garbage Collector zurück.

Es ist noch offen, wie der Schwellenwert festgelegt wird. Dazu werden von Chang und Huang zunächst Formeln hergeleitet, die das Verhältnis zwischen dem durch das Wear Leveling erzeugten Overhead und der Gleichmäßigkeit der Abnutzung aller Flashblöcke annähern sollen. Details dazu sind in [13], Abschnitt 4.1 veröffentlicht. Dort wird festgestellt, dass der Overhead nicht-linear ist und eine nur kleine Verringerung der Gleichmäßigkeit der Abnutzung aller Flashblöcke zu einem starken Overhead-Anstieg führen kann.

Lazy berechnet über daraus abgeleitete Formeln für bestimmte Zeitperioden, wie stark der Overheads für einen bestimmten Schwellenwert ansteigen würde, und passt

ihn automatisch so an, dass der Oberhead nur in geringem Maße vom vorherigen Wert abweicht (vgl. [13], Abschnitt 4.2 und Figure 7).

Diese Vorgehensweise bietet den Vorteil, dass keine zusätzlichen Informationen zwischengespeichert werden müssen, z.B. welche Daten gerade als *hot* oder *cold data* gelten. Lazy nutzt dazu einfach die implizit schon vorhandenen Informationen der Adressübersetzung.

Für einen Leistungsvergleich wurden unterschiedliche Echtsysteme mit verschiedenen Dateisystemen (z.B. NTFS, ext4, FAT32) simuliert (vgl. [13], Abschnitt 5.1 und Table 1). Das Ergebnis ist in Abbildung 14 dargestellt. Als Vergleich wurden zum einen die Einstellung „kein Wear Leveling“ und ein statisches Wear Leveling gewählt (ähnlich Dual Pool, s. letzter Abschnitt). Es zeigt sich, dass der zusätzliche Overhead, der durch Lazy oder das statische Wear Leveling erzeugt wird, nahezu gleichauf liegt mit einem Controller ganz ohne Wear Leveling. Es werden also nur wenige zusätzliche Blöcke wegen des Wear Levelings gelöscht, wie in Abbildung 14 (b) gezeigt. Der Test ohne Wear Leveling erreicht logischerweise das beste Ergebnis, da er nie Blöcke zusätzlich für Wear Leveling löschen muss. Auch die Standardabweichung, also das Maß der Gleichmäßigkeit der Abnutzung zeigt in Abbildung 14 (a) gute Ergebnisse.

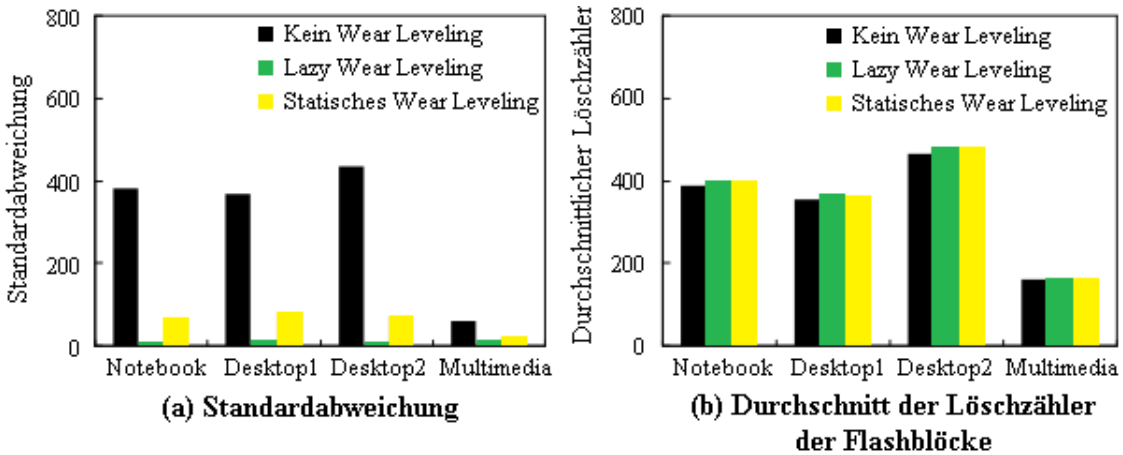


Abbildung 14: Performancevergleich Lazy Wear Leveling (vgl. [13], Figure 9)

Der Algorithmus wurde zum Test auch in eine SSD implementiert. Dort zeigten sich prinzipiell die gleichen Ergebnisse wie in Abbildung 14 gezeigt, bei einer Verrin-

gerung der I/O-Zugriffsgeschwindigkeit von knapp 3% gegenüber einem Controller ohne Wear Leveling (vgl. [13], Table 2).

Es zeigen sich keine großen Verbesserungen gegenüber dem statischen Wear Leveling, allerdings hat Lazy den großen Vorteil, dass es keine eigenen Datenverwaltungsstrukturen im Hauptspeicher benötigt, und daher besonders ressourcenschonend ist, aber dennoch das bislang beste Ergebnis erreicht.

4.2.1.5 Zukunft

In Zukunft könnte dieser Bereich möglicherweise keine Rolle mehr spielen: im Versuchsstadium ist es Wissenschaftlern gelungen, durch eine starke Erhitzung (ca. 800°C) der Floating Gate Transistoren die Abnutzung zu eliminieren (vgl. [52]). In Labortests konnten Zellen so schon über 100.000.000 Schreibzyklen überstehen.

4.3 Fehlerkorrektur

Um Fehler in Flashspeicher zu erkennen und beseitigen, werden Fehlerkorrekturverfahren eingesetzt. Es können viele Fehler erkannt, manche auch direkt behoben werden. Dies ist insbesondere bei MLC-NAND-Flash notwendig, da die Technik in diesen am anfälligsten für kleinere Fehler ist (vgl. Kapitel 2.4).

Der Fokus dieser Arbeit liegt jedoch überwiegend auf den anderen Flash-Mechanismen, da Fehlererkennung und -korrektur ein in vielen Technologien notwendiger Vorgang ist und sich auch bei Flashspeicher nicht grundlegend unterscheidet.

Eine grobe Übersicht über vorhandene Fehlerkorrekturverfahren und deren jeweilige Fähigkeiten, kann in der Veröffentlichung von Wang/Karpovsky/Joshi gefunden werden (vgl. [53], Abschnitt I und Table 2).

4.4 Adressübersetzung

4.4.1 Allgemeines

Jeder Flash-Block und jede Flash-Seite hat eine logische und eine physikalische Adresse. Die logische Adresse dient dazu, die referenzierten Daten immer auffindbar zu halten, selbst wenn sich die physikalische Adresse geändert haben sollte. Solche Änderun-

gen der physikalischen Adressen kann bei Flashspeicher häufig vorkommen, hauptsächlich durch die Garbage Collection und das Wear Leveling. Da Flash-Dateisysteme in der Regel ohne Umwege über einen Controller auf die Flash-Hardware zugreifen, ist dort nicht zwingend eine zusätzliche Adressübersetzung notwendig, da Änderungen des physikalischen Speicherorts direkt im Index des Dateisystems angepasst werden können.

Diese Übersetzungsmechanismen sind daher überwiegend für Flash Translation Layer interessant. Da manche Flashdateisysteme auf Flashspeicher mit Übersetzungs-Zwischenschicht arbeiten (z.B. F2FS), ist ein Verständnis der Funktionsweise der Adressübersetzung notwendig. Es werden daher an dieser Stelle einige ausgewählte Übersetzungsmechanismen vorgestellt, die in Controllern eingesetzt werden können. Die von den Herstellern tatsächlich eingesetzten Firmwares gelten jedoch als Geschäftsgeheimnisse Daher ist nahezu nichts über die internen Funktionsweisen bekannt (vgl. [54], Folie 8 und [55], Seite 1).

Zur Laufzeit sollte die Zuordnung in sehr schnellem Speicher vorgehalten werden, damit kein Zeitverlust bei der Übersetzung entsteht.

Prinzipiell gibt es drei Möglichkeiten der Übersetzung:

1. auf Seiten-Ebene („page-level“)
2. auf Block-Ebene („block-level“)
3. und Hybrid-Verfahren

4.4.1.1 Seitenweise Übersetzung

Bei seitenweiser Übersetzung (Variante 1) wir ein logischer Sektor des Betriebssystem-Dateisystems auf eine logische Seite des Flash-Speichers abgebildet. Dadurch kann die Garbage Collection sehr effizient durchgeführt werden, jedoch hat es den Nachteil, dass viel RAM für die Zuordnungstabelle benötigt wird (vgl. [56]). Wenn beispielsweise in Sektor „9“ der Inhalt „A“ geschrieben werden soll, funktioniert die Übersetzung wie in Abbildung 15 gezeigt.

Je nachdem wie feinkörnig die Flash-Seiten sind (z.B. 512 Byte oder 4 KByte pro Seite), entsteht dadurch ein unterschiedlich hoher Platzbedarf für die Übersetzungsta-

belle. Daher wäre eine hohe Seitengröße wünschenswert, um möglichst wenig RAM für die Tabelle zu benötigen. Andererseits ist die Flash-Seite die kleinste Einheit, auf die schreibend zugegriffen werden kann. Ist eine hohe Seitengröße gegeben, wird bei kleinen Änderungen dennoch immer die ganze Seite neu geschrieben. Daher wäre für die Performance und Lebensdauer des Flash-Speichers eine eher kleine Seitengröße wünschenswert.

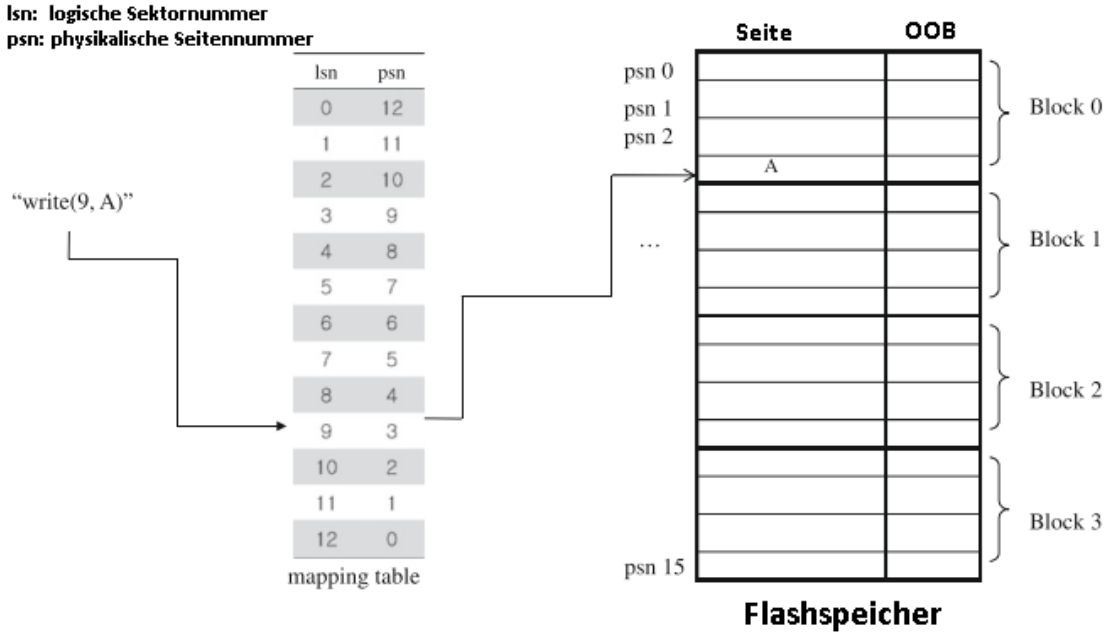


Abbildung 15: Adressübersetzung auf Seiten-Ebene (vgl. [88], S. 334)

4.4.1.2 Übersetzung auf Block-Ebene

In der zweiten Variante wird jedem physikalischen Block eine logische Blocknummer zugewiesen. Um eine logische Seite des Dateisystems zu beschreiben, wird somit die logische Blocknummer und ein Offset (für die Seite innerhalb des Blocks) benötigt. Dadurch wird der Hauptspeicherbedarf erheblich verringert, da nur eine Übersetzungstabelle für die (relativ großen) Blöcke vorgehalten werden muss. Wenn jedoch nur eine Seite in einem Block verändert werden muss, so muss der gesamte Block neu geschrieben werden: es wird der die neu zu schreibende Seite in einen neuen Block geschrieben, danach alle anderen Seiten des Blocks in den neuen Block kopiert und schließlich der alte Block gelöscht. Dadurch werden häufig Blöcke gelöscht und neu geschrieben, selbst wenn in ihnen nur wenige Seiten genutzt wurden (vgl. [57], S.

753). Auch im Hinblick auf die begrenzten Löschzyklen, die ein Flashblock durchleben kann, ist dies problematisch. Für den Fall, dass jeder Block genau vier Seiten beinhaltet, funktioniert die Adressübersetzung auf Blockebene wie in Abbildung 16 gezeigt. Es wird wieder davon ausgegangen, dass ein Schreibbefehl auf Sektor 9 mit dem Inhalt „A“ ausgeführt werden soll. Der Offset ergibt sich aus dem Sektor modulo der Anzahl der Seiten pro Block.

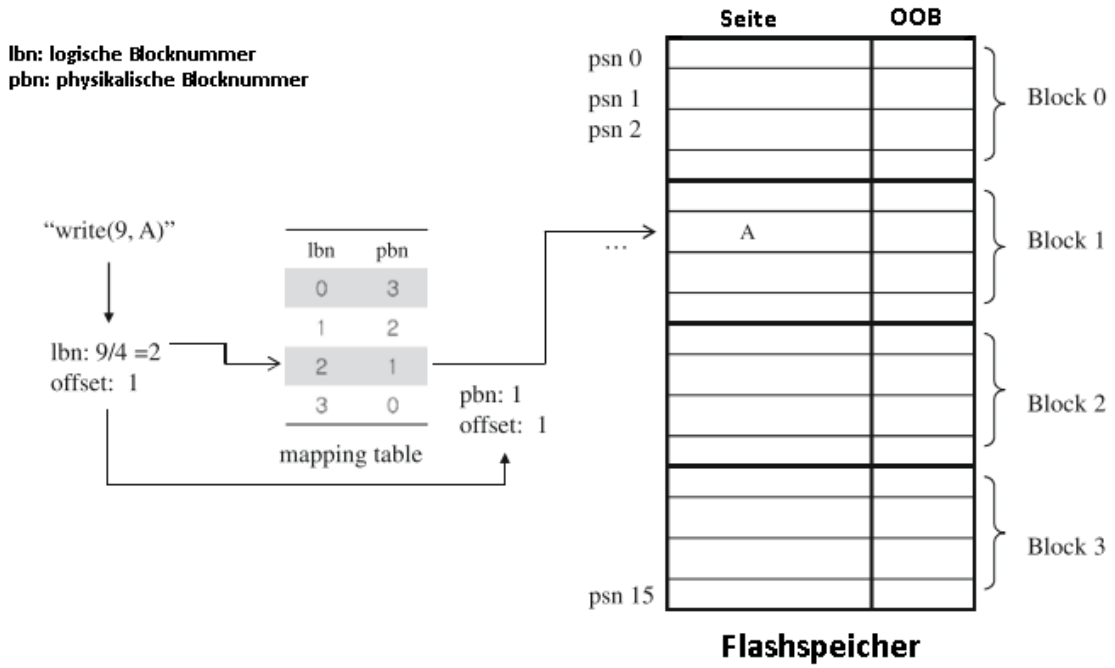


Abbildung 16: Adressübersetzung auf Block-Ebene (vgl. [88] S. 334)

4.4.1.3 Hybrid-Verfahren

Um die genannten Nachteile zu vermeiden, wurden Hybrid-Verfahren entwickelt. Diese haben nur einen geringen Garbage Collection Overhead und benötigen dennoch nur wenig Hauptspeicher für die Zuordnungstabelle, indem sie grundsätzlich das Mapping auf Block-Ebene durchführen, die einzelnen Seiten innerhalb eines Blocks jedoch in einer zufälligen Reihenfolge gespeichert sind. Bei Schreibvorgängen auf den Block wird unabhängig vom Offset einfach die nächste freie Seite benutzt. Um eine bestimmte Seite später identifizieren zu können, wird das Offset der Seite im Out-Of-Band-Bereich (OOB) zu dieser Seite gespeichert. Hier wird zur Bezeichnung der Speicherstelle für Schreib-/Leseoperationen die Blocknummer und die logische Sektornummer

genutzt. Um die Daten eines Sektors zu lesen, müssen die OOB-Bereiche der Seiten des Blocks so lange gelesen werden, bis der richtige Sektor gefunden wurde. Dies führt zu einem geringen RAM-Bedarf und dazu, dass nicht ein gesamter Block neu geschrieben werden muss, wenn sich nur eine Seite darin ändert. Die prinzipielle Funktionsweise von Hybrid-Übersetzungsverfahren ist in Abbildung 17 gezeigt.

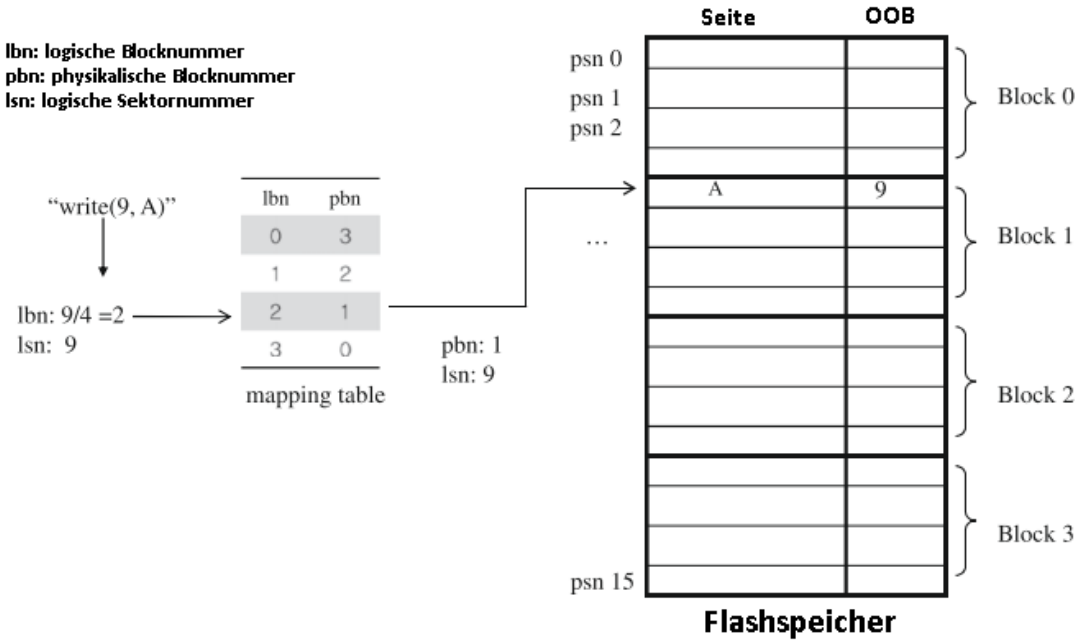


Abbildung 17: Adressübersetzung von Hybrid-Verfahren (vgl. [88] S. 335)

Die Übersetzung rein auf Block- oder Seitenebene wird wegen der genannten Nachteile kaum noch eingesetzt, jedoch werden bei den Hybrid-Verfahren immer wieder neue Vorgehensweisen vorgeschlagen. Im Folgenden ein Überblick darüber.

4.4.2 Hybrid-Übersetzungsverfahren

Repräsentativ werden im Folgenden einige Hybrid-Übersetzungsverfahren dargestellt.

4.4.2.1 Block-associative sector translation (BAST, 2002)

BAST[22] ist eines der Hybrid-Verfahren. Die Übersetzung funktioniert grundlegend auf Block-Ebene – die weit überwiegende Zahl aller Blöcke wird so angesprochen, in BAST werden sie einfach „*data blocks*“ genannt. Allerdings werden systemweit einige

[22] In manchen Veröffentlichungen wird BAST auch als „log block scheme“ bezeichnet

zusätzliche Blöcke geführt, die „*log blocks*“ und „*map blocks*“ genannt werden (vgl. [58], S. 368).[23]

Log blocks sind feinkörniger als *data blocks* und auf Seitenebene adressierbar. Sie werden immer genau einem *data block* zugeordnet. Kleine Schreiboperationen werden anstatt auf den *data block* in den zugehörigen *log block* geschrieben. Die logische Adresse der überschriebenen Daten des *data blocks* wird im „Out of Band“-Bereich der Seite des *log blocks* vermerkt. Bei Lese-Operationen auf eine Adresse im *data block* muss allerdings zuerst der *log block* durchsucht werden, ob dort eine aktuellere Version der angeforderten Daten vorhanden ist und gegebenenfalls die Daten von dort gelesen werden. Damit dies effizient geschieht, wird die Mapping-Tabelle für *log blocks* in RAM gespeichert. Diese Mapping-Tabelle wird beim Systemstart aufgebaut, indem alle Out-of-Band-Bereiche gelesen werden, in denen die Mapping-Informationen abgelegt sind.

Wenn einem *data block* ein *log block* zugeordnet wurde, werden alle Schreiboperationen auf den *log block* ausgeführt, bis dieser voll ist. Dabei können Daten im *data block* auch mehrfach „überschrieben“ werden, was zu mehreren Einträgen im *log block* führt. Wenn der *log block* voll ist oder ein freier *log block* benötigt wird, wird eine Merge-Operation[24] durchgeführt, bei der der *data block* und der *log block* verschmolzen werden und das Ergebnis ein einziger, neuer *data block* ist. Die zwei verschiedenen Möglichkeiten werden in Abbildung 18 dargestellt.

[23] Auch spätere Übersetzungsverfahren verwenden die beiden Blocktypen *data block* und *log block*. Es haben sich jedoch andere Bezeichnungen dafür durchgesetzt: so wird der „*log block*“ in anderen Verfahren als U-Block („update block“) bezeichnet und der „*data block*“ als D-Block („data block“)

[24] Engl. „to merge“: zusammenführen, verschmelzen

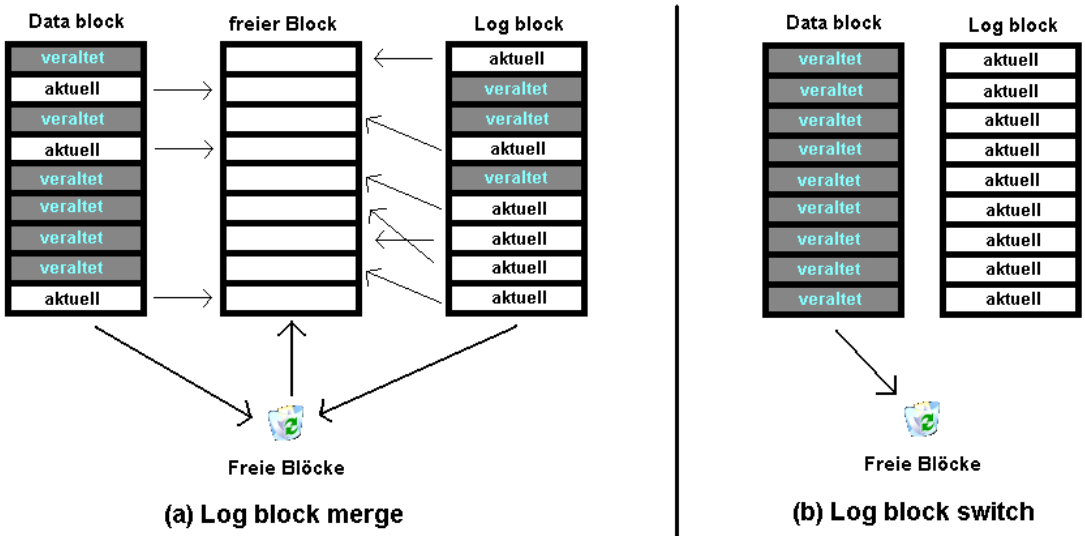


Abbildung 18: Möglichkeiten der Zusammenführung von data block und log block

In der ersten Variante (a) werden die noch gültigen/aktuellen Daten aus dem *data block* mit den aktuellen Daten aus dem *log block* in einen neuen, freien Block zusammengeführt. Der zusammengeführte Block ist danach der neue *data block*, der alte *data block* und *log block* können gelöscht werden und werden dann der Liste der freien Blöcke zugeordnet.

Eine zweite Variante (b) kann auftreten, wenn alle Seiten eines Blockes sequentiell von der ersten bis zur letzten Seite neu geschrieben werden. Dann kann der alte *data block* direkt gelöscht werden und der zugehörige *log block* ist der neue *data block*. Dieser Vorgang wird auch „log block switch“ oder „switch merge“ genannt. Üblicherweise tritt er wesentlich seltener auf und ist sehr viel schneller als Variante (a), die auch als „full merge“ bezeichnet wird.

Map blocks dienen dazu, um einerseits die Mapping-Tabelle beim Systemstart schneller aufbauen zu können und andererseits zur Sicherheit, damit auch bei Stromausfällen die Mapping-Informationen nicht verloren gehen.

Vor BAST wurden die Mapping-Informationen bei jedem Systemstart erneut aufgebaut und im RAM gehalten, indem alle Out-of-Band-Bereiche des gesamten Flash-Speichers gelesen wurden. Dies ist allerdings bei schnell größer werdenden Speichermedien sehr zeitaufwändig. Daher wird in BAST die gesamte Mapping-Tabelle in *map blocks* gespeichert, die genau wie *log blocks* auf Seitenebene adressierbar sind.

Im SRAM selbst sind nur noch die Adressen der *map blocks* abgelegt (sog. „*map directory*“). Das Verfahren funktioniert daher so ähnlich wie die indirekte Adressierung über Inodes in ext2. Jedes Mal wenn ein Block eine neue Adresse erhält, wird daher ein neuer Eintrag in einen *map block* geschrieben. Daher werden z.B. bei dem „Log Block Merge“-Vorgang (Abb. 18a) drei Einträge generiert: der neue Block erhält die Adresse des ursprünglichen *data blocks*, der alte *data block* und der *log block* werden zur Liste der freien Blocks hinzugefügt. In jede Seite eines *map blocks* können mehrere Blockadressen gleichzeitig abgelegt sein – das hat den Vorteil, dass Schreibvorgänge atomar sind und auch bei Systemabstürzen oder Stromausfällen die Integrität der Adressinformationen bestehen bleibt.

Die *map blocks* werden durch Änderungen irgendwann voll geschrieben sein. Dann müssen sie, wie auch bei der Garbage Collection, gesäubert und zusammengefasst werden. In BAST wird dafür ein Round-Robin-Verfahren für die *map blocks* verwendet und immer der nächste nach dem aktuell verwendeten *map block* gesäubert, d.h. seine noch gültigen Daten in den aktuellen Block kopiert, bevor er gelöscht und wiederverwendet wird.

Beim Systemstart werden die *map blocks*, die sich an starr festgelegten Stellen befinden, gescannt um das *map directory* im RAM aufzubauen. Gerade wegen der starren Stellen, an denen *map blocks* gespeichert sind, kann an sich an dieser Stelle jedoch ein Problem mit der maximalen Anzahl an Löschvorgängen auf diesen Blocks ergeben. Daher funktioniert die Adressauflösung in zwei Stufen: die höherwertigen Bits der Adresse identifizieren über das *map directory* den zugehörigen *map block*, in welchem die Adresse der eigentlich gesuchte Speicherblocks steht. Dort kann dann mit dem Offset die zu lesende Information gefunden werden.

BAST war die erste Umsetzung einer Hybrid-Strategie, die geringe Hauptspeicheranforderungen erfüllt und dennoch bei kleinen Schreibbefehlen nicht ganze Blocks neu schreibt. Dieser Ansatz kann an vielen Stellen noch verbessert werden, z.B. bieten sich Optimierungsmöglichkeiten für Merge-Operationen, so dass diese im Hintergrund bei geringer I/O-Auslastung stattfinden könnten. Auch wird bei jeder Schreiboperation auf einen data block ein *log block* benötigt, was zu dem Problem führe kann, dass es viele

kaum gefüllte, ineffiziente *log blocks* gibt (vgl. [59], S. 164). Damit wurde aber ein Anfang für die folgenden Algorithmen geschaffen.

4.4.2.2 Fully-Associative sector translation (FAST, 2006)

Lee et al. haben dieses Adress-Übersetzungsverfahren bereits 2006 entwickelt, es wurde 2007 veröffentlicht (vgl. [60]). Dort wurde festgestellt, dass das zu diesem Zeitpunkt beste Übersetzungsverfahren, BAST, immer noch sehr häufig Blöcke löschen muss: der Grund dafür liegt darin, dass jeder *log block* nur genau zu einem *data block* gehören kann, aber systemweit immer nur eine geringe Anzahl an *log blocks* vorhanden ist. Sobald ein neuer *log block* benötigt wird und kein freier verfügbar ist, muss einer der alten *log blocks* mit seinem *data block* zusammengeführt und dann gelöscht werden. Die *log blocks* sind häufig aber nur mit wenigen gültigen Seiten gefüllt. Das ist demnach insbesondere dann ein Problem, wenn viele zufällige Schreibzugriffe auftreten.

Das will das FAST-Verfahren ändern. Hier können die Daten, die in einem *data block* überschrieben werden sollen, in jeden beliebigen *log block* geschrieben werden. Die *log blocks* in FAST sind keinem bestimmten *data block* zugeordnet. Dadurch werden viele Merge-Operationen verzögert, bis auch in FAST keine Seite mehr in einem *log block* verfügbar ist. Manche werden ganz ausgelassen, da möglicherweise die Daten, die sonst zusammengefasst worden wären, bereits erneut überschrieben wurden. Weiterhin werden teure Merge-Operationen mit fast leeren *log blocks* komplett vermieden.

Weiterhin wird in FAST zwischen zwei verschiedenen *log blocks* unterschieden: einen für zufällige Schreibzugriffe und einen für sequentielle Schreibzugriffe. Der Grund dafür liegt in den verschiedenen möglichen Zusammenführungs-Operationen (siehe Abbildung 18). Die Switch-Operation ist deutlich effizienter als Merge-Operation. Wenn sequentielle und zufällige Schreibzugriffe jedoch vermischt würden, könnte FAST wegen seines Ansatzes, jeden *log block* für alle *data blocks* zu verwenden, kaum Switch-Operationen durchführen.

Um in den *log block* für sequentielle Schreibzugriffe geschrieben zu werden, müssen die zu schreibenden Daten eine von zwei Bedingungen erfüllen:

1. Die Seite hat einen Offset von 0 in ihrem Block oder
2. im sequentiellen *log block* sind bereits alle vorhergehenden Seiten des zugehörigen Blocks vorhanden, so dass der aktuelle Schreibzugriff die Reihe fortführen würde.

Ist keine der beiden Bedingungen erfüllt, wird die veränderte Seite in den *log block* für zufällige Schreibzugriffe gespeichert (mehr Details dazu in [60], Abschnitt 3.3).

Wie im BAST-Verfahren muss auch LAST bei Lesezugriffen auf eine bestimmte Seite überprüfen, ob in einem der log blocks eine neuere Version diese Seite vorhanden ist. Hier entsteht bei LAST ein etwas größerer Overhead, da eine aktualisierte Seite in deutliche mehr log blocks gefunden werden könnte, als es in BAST der Fall war. Dies wird durch eine Mapping-Tabelle im RAM gelöst, in der ein solcher Suchvorgang im Durchschnitt ca. 1µs dauert; das fällt kaum ins Gewicht, wenn bedacht wird, dass das Lesen einer Flash-Seite typischerweise 15 µs dauert.

Meta-Daten werden in FAST genau wie in BAST über *map blocks* verwaltet und zusätzlich im Out-Of-Band-Bereich der Seiten gespeichert. Die Vorgehensweise ist in beiden Verfahren identisch.

Sobald keine Seite in einem *log block* mehr verfügbar ist, wird im Round-Robin-Verfahren ein *log block* als Opferblock ausgewählt.

Wenn ein voller *log block* vorhanden ist, der sequentielle Zugriffe gespeichert hatte, kann einfach ein *log block* switch (vgl. Abb. 18b), genau wie in BAST, durchgeführt werden. Für die *log blocks*, die zufällige Schreibzugriffe gespeichert haben, ist die Vorgehensweise anders:

In FAST können Seiten verschiedener *data blocks* im selben *log block* gespeichert sein. Daher werden alle Seiten im *log block*, die zum gleichen *data block* gehören, jeweils in leere Blöcke kopiert. Danach werden aus den alten *data blocks* die noch fehlenden Seite ebenfalls in den leeren Blöcke ergänzt. Nun können sowohl der *log block* als auch die alten *data blocks* gelöscht werden.

Es wird in der gleichen Veröffentlichung (vgl. [60], Abschnitt 3.6) noch eine Verbesserung namens optimized FAST (O-FAST) vorgestellt. Diese versucht, Merge-Operationen so lange wie möglich zu verzögern. Dort werden bei der Merge-Operation, die im vorherigen Absatz beschrieben wird, bei der Garbage Collection eines *log blocks* nur noch solche Seiten in leere Blöcke kopiert, für die es noch keine neuere Version in anderen *log blocks* gibt. Dadurch werden keine Merges mit Daten durchgeführt, die zu dem Zeitpunkt bereits veraltet waren.

Der Vergleich zwischen BAST, FAST und O-FAST wurde mit einigen verschiedenen Workloads getestet. In den folgenden Abbildungen sind beispielhaft einige davon

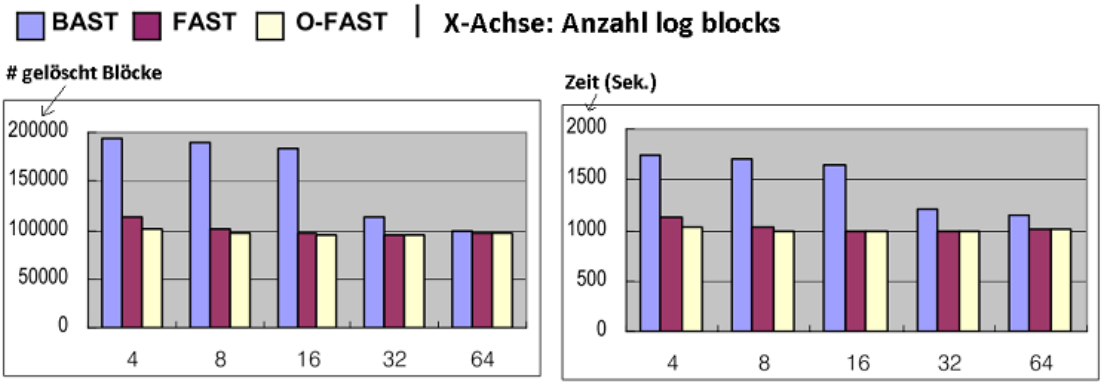


Abbildung 19: Performance-Vergleich FAST: Digitalkamera (vgl. [60] Abschnitt 4)

gezeigt.

Abbildung 19 zeigt den Vergleich für den Workload eine Digitalkamera mit insgesamt ca. 2.2 Millionen Schreibzugriffen, darunter zufällige und große sequentielle. Gerade wenn systemweit nur wenige *log blocks* eingesetzt werden, zeigt sich ein deutlicher Vorsprung vor FAST.

Der nächste Vergleich zeigt einen Workload unter Linux mit ca. 400.000 Schreibzugriffen, auch hier wieder zufällige und sequentielle (vgl. Abbildung 20). Hier ist der Vorsprung von FAST gegenüber BAST noch vorhanden, allerdings nicht sehr groß.

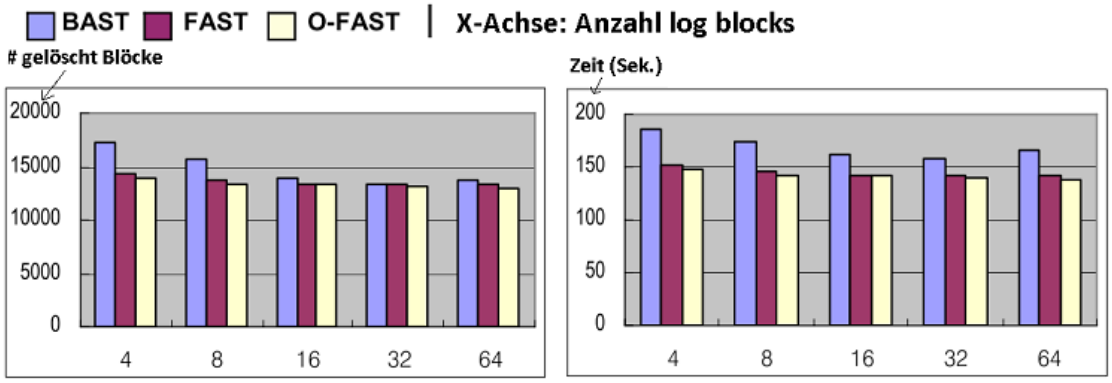


Abbildung 20: Performance-Vergleich FAST: Linux (vgl. [60], Abschnitt 4)

Und zuletzt noch ein Vergleich zu einem künstlich erzeugten Workload aus 150.000 zufälligen Schreibzugriffen (vgl. Abbildung 21). Hier wird besonders deutlich, dass die Stärke von FAST gerade mit einem solchen Workload ausgespielt werden kann.

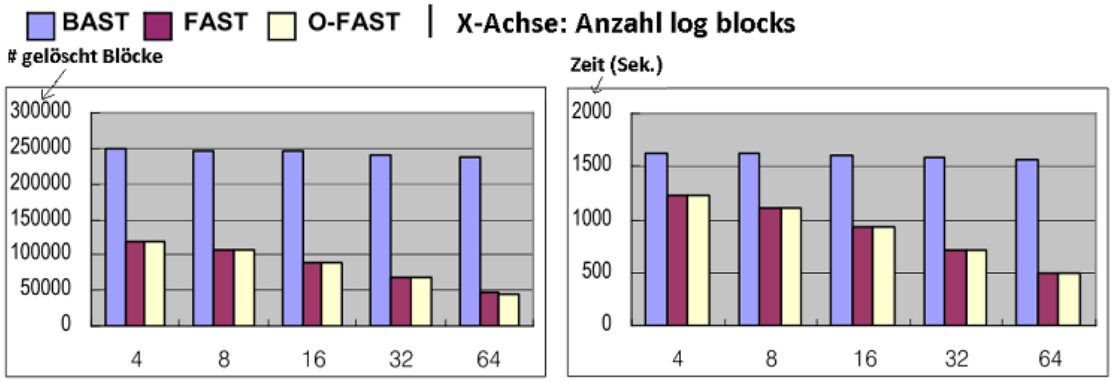


Abbildung 21: Performance-Vergleich FAST: Zufällige Schreibzugriffe (vgl. [60], Abschnitt 4)

Das FAST-Adressübersetzungsverfahren war damit dem bisher besten Verfahren, BAST, in jeder Hinsicht überlegen. Gerade bei vielen zufälligen Schreibzugriffen zeigt sich eine enorme Verbesserung.

4.4.2.3 Locality-Aware sector translation (LAST, 2008)

Die Autoren des LAST-Papers (vgl. [56]) haben einige Schwächen in BAST, FAST und in Superblock (vgl. [59], hier nicht vorgestellt) entdeckt und wollen diese mit dem LAST-Verfahren überwinden. BAST und FAST funktionieren gut in spezieller Consumer-Hardware (z.B. Digitalkameras, MP3-Player, etc.), die überwiegend sequentielle

Schreibzugriffe in einem Write-Stream durchführt, sind bei zufälligen Schreibvorgängen jedoch langsam. Das will FAST zwar durch einen log block für sequentielle Zugriffe verhindern, jedoch gibt es bei Computersystemen wie etwa PCs und Notebooks öfter mal mehrere Schreib-Ströme gleichzeitig, die mit sequentiellen Zugriffen vermischt werden und damit das FAST-System wieder aushebeln. Superblock ist dort zwar besser, hat aber andere Schwächen: zum einen wird die Unterscheidung zwischen *hot* und *cold data* als ineffizient bezeichnet, zum anderen wird die Metadaten-Verwaltung mit dem Mapping innerhalb der Superblöcke als zu aufwändig in der Verwaltung angesehen (vgl. [56], S. 38).

Die Architektur von LAST ist in Abbildung 22 dargestellt. Wenn Schreibzugriffe (oben, „Write Requests“) ankommen, soll der Lokalitätsdetektor („Locality Detector“) feststellen, ob es sich um einen zufälligen oder einen sequentiellen Schreibzugriff handelt. Wird ein sequentieller Zugriff festgestellt, werden die Daten in einen von mehreren log buffern[25] geschrieben, der explizit dafür gedacht ist und dessen Funktionsweise mit der BAST-Architektur identisch ist: jeder solche Block wird explizit dem zugehörigen *data block* zugeordnet. Bei zufälligen Schreibzugriffen wird das Verfahren genutzt, das FAST verwendet: es wird in *log blocks* geschrieben, die von *data blocks* unabhängig sind. Zusätzlich wird noch unterschieden zwischen *log blocks* für *hot data* und *log blocks* für *cold data* (hot partition/cold partition): damit ist in LAST gemeint, dass eine hohe zeitliche Lokalität besteht, also häufig in einem kurzen Zeitraum auf die gleichen Daten zugegriffen wird.

Die Merge-Operation wird wie schon bei den anderen Verfahren dann durchgeführt, wenn es keine *log blocks* mehr gibt, in die neue Schreibzugriffe geschrieben werden können.

[25] Als „log buffer“ wird einer oder mehrere log blocks bezeichnet, die für den vorgegebenen Zweck (z.B. sequentielle/zufällige Zugriffe) gedacht sind

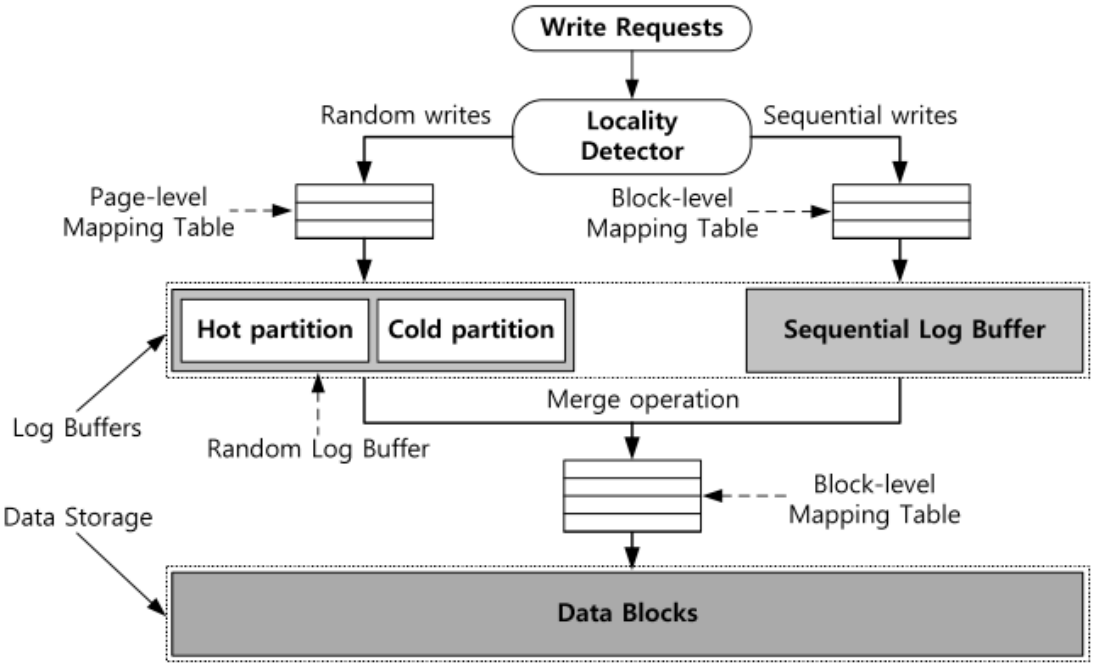


Abbildung 22: LAST-Architektur (vgl. [56], S. 38)

Der Lokalitätsdetektor wird relativ einfach konfiguriert: alle Schreibzugriffe, deren Größe einen bestimmten einzustellenden Schwellenwert überschreitet (in der Veröffentlichung: 4 KB), werden in den sequentiellen log buffer geleitet. Alle kleineren Schreibzugriffe werden in den log buffer für zufällige Zugriffe geschrieben. Dieser Wert wurde empirisch aus dem Zugriffsverhalten eines Universalcomputers[26] ermittelt: es wurde festgestellt, dass bei nahezu allen größeren Schreibzugriffen immer mehrere zusammenhängende Sektoren gleichzeitig an die FTL zum Überschreiben übergeben wurden, während bei zufälligen Zugriffen meist nur eine geringe Sektoren-Zahl beschrieben werden sollte.

Um abzuleiten, wie LAST mit den log buffern für zufällige Schreibzugriffe umgehen soll, wurde zunächst mathematisch gezeigt, wie hoch die Kosten[27] für eine Merge-Operation sind:

$$Kosten\,der\,Merge-Operation = Migrationskosten + Löschkosten$$

$$Kosten\,der\,Merge-Operation = N_a \cdot ((N_p \cdot C_c) + C_e) + C_e$$

Die Kosten setzen sich zusammen aus der Summe Migrationskosten, also dem Zusammenführen noch gültiger Seiten mit dem zugehörigen *data block*, sowie den

[26] Eines „general purpose computers“, gemeint ist ein PC oder Notebook
[27] Hierbei wird rein auf die Zeitkosten eingegangen; Wear Leveling ist nicht berücksichtigt

Löschkosten für dann nicht mehr benötigte Blöcke. Da in einem *log block* viele Seiten vorhanden sein können, die zu vielen unterschiedlichen *data blocks* gehören können, wird der Begriff „Assoziativitätsgrad“ eingeführt. In der Formel ist dies N_a, das angibt, wie viele *data blocks* mit dem *log block* in Beziehung stehen. Alle gültigen Seiten eines jeden assoziierten *data blocks* müssen in einen neuen, leeren Block kopiert werden; die anderen Seiten werden aus den *log blocks* übernommen. N_p bezeichnet die Anzahl an Seiten pro Block und C_c die Kopier-Kosten pro Seite. Schließlich steht C_e für die Löschkosten eines Blocks.

Es wird hier deutlich, dass der Assoziativitätsgrad eine sehr starke Auswirkung auf die Kosten einer Merge-Operation hat und ein Algorithmus insbesondere darauf Wert legen sollte, ihn zu verringern. LAST will das erreichen, indem es die zeitliche Lokalität ausnutzt. Dies wird zum einen über die Trennung in *hot* und *cold data* erreicht, indem *hot data* in die gleichen *log blocks* geschrieben wird. *Hot data* wird häufig geändert, daher werden viele Seiten im *log buffer* schnell wieder ungültig, häufig sogar alle Seiten des *log blocks*, so dass er nur noch gelöscht werden muss.[28] Zum anderen wird gewartet, bis der Assoziativitätsgrad sich durch häufige Änderungen von selbst verringert. Da dies bei cold pages viel seltener der Fall sein wird, wird bevorzugt ein Opferblock aus dem cold page log buffer ausgewählt.

Die Anzahl an *log blocks* für *hot* und *cold data* wird dynamisch ermittelt. Die Klassifizierung nach hot/cold basiert darauf, ob eine bestimmte Seite innerhalb der letzten *k* Schreibzugriffe mehr als einmal vorkommt. Ist dies der Fall, gilt diese Seite als *hot*, sonst als *cold*. Auch k wird dynamisch angepasst. Weitere Details können bei Lee et al. nachgelesen werden (vgl. [56], S. 40).

Wenn im log buffer für zufällige Schreibzugriffe ein neuer Block benötigt wird, sucht der Algorithmus nahezu immer den Block mit den geringsten Merge-Kosten. Einzig wenn ein log block in der hot partition seit einem zu definierenden Schwellenwert nicht mehr verändert wurde, wird dieser gewählt, wenn kein dead block vorhanden ist.

Im Leistungsvergleich mit echten Traces von Computersystemen zeigt sich eine deutliche Verbesserung von LAST gegenüber anderen Adressübersetzungsverfahren

[28] Ein log block der nur noch ungültige Seiten enthält wird auch als „dead block“ bezeichnet

(Abbildung 23).[29] Es wurden 32 GB Flashspeicher genutzt mit einer log buffer Größe von 512 MB.

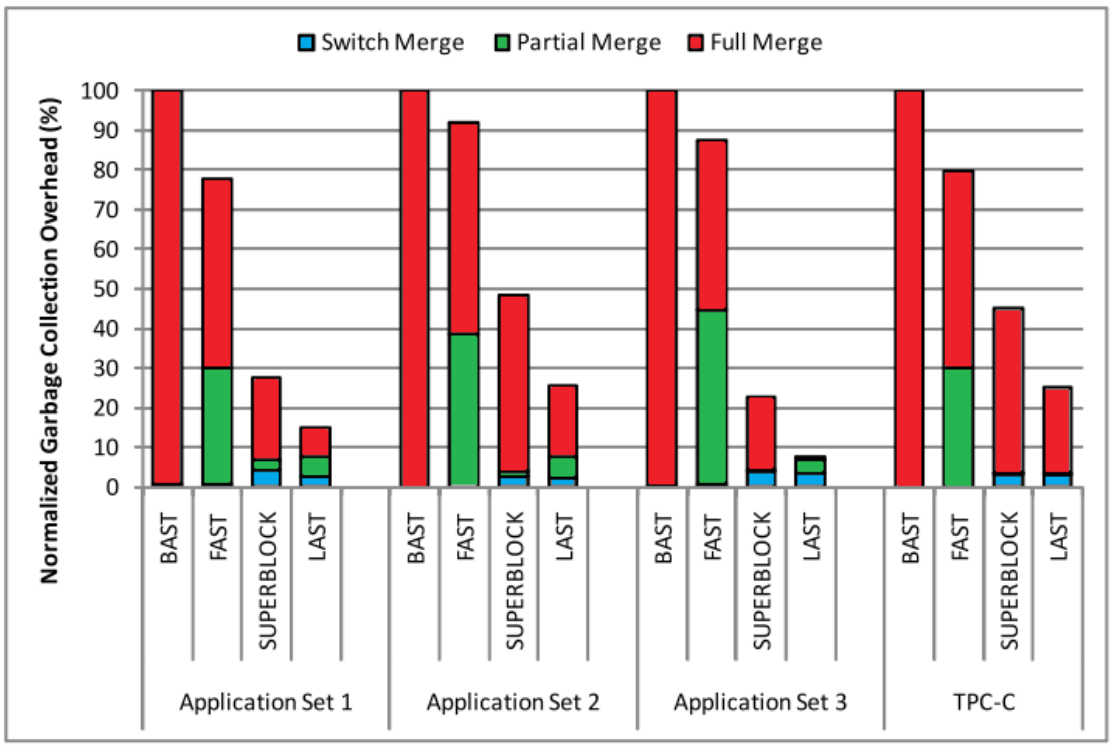


Abbildung 23: LAST-Leistungsvergleich (vgl. [56], S. 41)

4.4.2.4 ROSE (2011)

Das wenige Jahre später veröffentlichte ROSE funktioniert prinzipiell genauso wie die anderen vorgestellten Hybrid-Übersetzungsverfahren, optimiert allerdings einige der Mechanismen (vgl. [57]).

Die erste der neuen Techniken, genannt *Entire Block Writing*, verhindert, dass die Daten bei sequentiellen Schreibzugriffen in unterschiedliche Blöcke geschrieben werden (vgl. [57], Abschnitt 3.1). Im Gegensatz zu LAST wird keine Vorhersage -Routine eingesetzt, die Schreibzugriffe in zufällig und sequentiell unterteilt, da diese gelegentlich falsche Ergebnisse geliefert hat, wodurch zusätzlicher Aufwand für die Garbage Collection erzeugt wird. Stattdessen wird jeder Schreibzugriff von ROSE intern direkt in blockweise und seitenweise Zugriffe zerteilt. Wenn also z.B. in einer Flasharchitek-

[29] Application Set 1 = Windows XP Notebook/PC; Application Set 2 = Windows XP Notebook/PC; Application Set 3 = Windows XP Notebook/PC; TPC-C = Oracle DB Enterprise Server System

tur mit 64 Seiten pro Block ein Schreibzugriff auf die logischen Seiten 0-140 ankommt, wird dieser automatisch in zwei blockweise Zugriffe umgesetzt (0-63 und 64-127) und in einen seitenweisen Zugriff (128-140). ROSE versucht demnach nicht sequentielle Zugriffe vorherzusagen, sondern erkennt sie.

Eine weitere Optimierung wurde bei der Auswahl des zu säubernden Blocks der Garbage Collection realisiert. Es wird die Idee von LAST erweitert. Wie dort werden die Gesamtkosten inklusive aller assoziierten anderen *data* und *log blocks* für die Garbage Collection betrachtet. Zusätzlich wird auch noch die Zeit mit einbezogen, die seit der letzten Säuberung dieses *log blocks* vergangen ist. Es wird erwartet, dass bei noch jungen *log blocks* in naher Zukunft weitere Seiten ungültig werden und die Säuberungskosten dann weiter fallen. Es ist im Prinzip die gleiche Grundidee, die auch hinter dem Cost-Benefit-Garbage Collector steckt (vgl. Kap. 4.1.2.2).

Es wird daraus ein Wert für jeden Block berechnet, der sich aus dem Alter des Blocks und den Säuberungskosten zusammensetzt. Die genaue Formel dazu ist komplex und in der Veröffentlichung dargestellt (vgl. [57], S. 759, Formel 5). Der Block, der den höchsten Wert erreicht, wird als victim block ausgewählt. Die Komplexität führt allerdings dazu, dass die Berechnung der Werte aller Blöcke zu lange dauern würde, wenn die Garbage Collection aktiv wird. Daher werden bei jedem Schreibzugriff auf eine Seite der Wert für alle davon betroffenen Blöcke neu berechnet (vgl. [57], S. 759). Auch den Block mit dem höchsten Wert zu finden ist aufwändig, weswegen der gesamte Flashspeicher dafür in Cluster unterteilt wird, die jeweils den größten Wert der beinhalteten Blöcke speichern. Sogar zusätzliche Hardwareschaltungen wurden zur Leistungssteigerung vorgesehen (vgl. [57], S. 759). Dadurch erreicht ROSE etwa die gleiche Berechnungskomplexität wie LAST.

Zuletzt verwendet ROSE freie Seiten von bereits ungültigen Blöcken. Dies stellt die dritte Optimierung dar. *Log blocks* werden immer bis zur letzten Seite geschrieben, dies gilt jedoch nicht für *data blocks*. Es kann z.B. passieren, dass ein *log block* von der Garbage Collection gesäubert werden soll und er damit via Merge-Operationen mit allen zugehörigen *data blocks* verschmolzen wird. Wenn einer der zugehörigen *data blocks* zu dem Zeitpunkt beispielsweise nur eine beschrieben Seite hatte, so wird diese

dennoch in einen neuen Block kopiert. Der alte *data block* ist nun ungültig, hat jedoch noch viele freie Seiten verfügbar. Diesen Block mit allen noch freien Seiten kann ROSE als neuen *log block* benutzen, ohne ihn vorher löschen zu müssen.

In Leistungstests hat sich ergeben, dass ROSE im Hinblick auf die Schreib- und Säuberungskosten jedem anderen Hybrid-Übersetzungsschema überlegen ist (vgl. Abbildung 24 und weitere Vergleiche in [57], Abschnitt 4). Dort wurden mehrere Traces unterschiedliche Geräte mit ROSE, LAST und FAST getestet.[30] Die Anzeige „ROSE w/o FPR“ bedeutet, dass ROSE in dem Fall ohne die Nutzung von freien Seiten ungültiger Blöcke getestet wurde, d.h. ohne die dritte Optimierung. Es zeigt sich, dass mit ROSE bei Schreibbefehlen durchweg nur wenige zusätzliche Schreibzugriffe notwendig waren.

Allerdings bestehen die Schreib- und Säuberungskosten nur aus den Lösch- und Kopierkosten, nicht jedoch dem Rechenzeitbedarf. Wie hoch der wäre, insbesondere ohne die Hardwareanpassungen, wird leider nirgends aufgeführt.

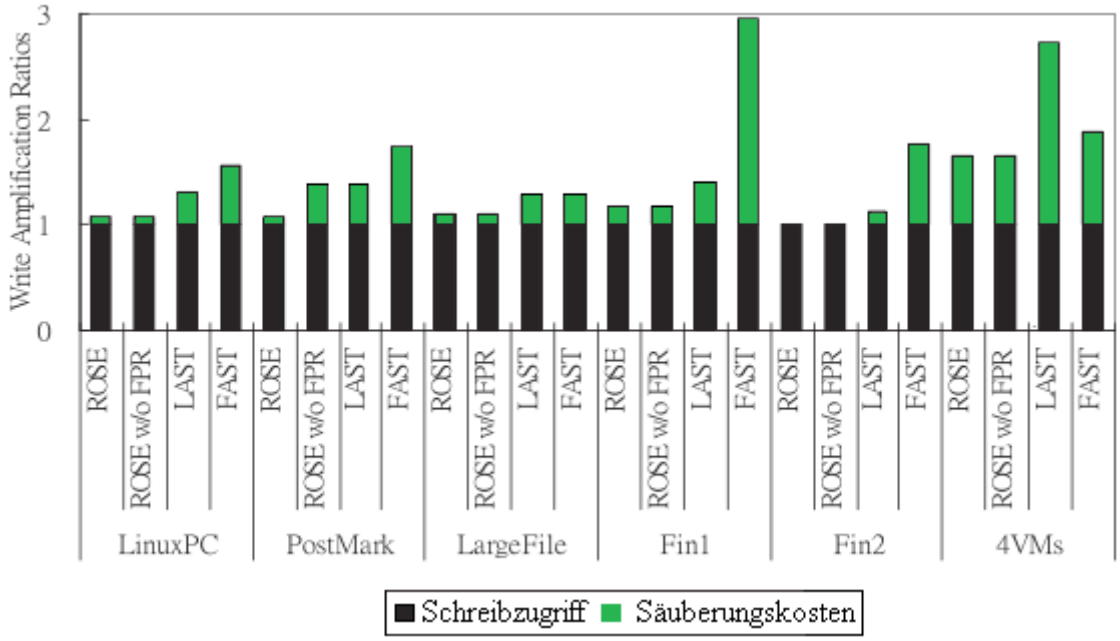


Abbildung 24: Vergleich der Schreibkosten von ROSE, LAST und FAST (vgl. [57], Fig. 14)

[30] LinuxPC = 10 Tage Aufzeichnung eines Benutzers; Postmark: Benchmark-Tool; LargeFile: Schreiben und Löschen von MP3-Dateien; Fin1 und 2 = Transaktionsverarbeitungsprogramme großer Finanzinstitute; 4 VMs = vier virtuelle Maschinen mit File-/Mail-/Proxy-Server und Transaktiosverarbeitungsprogramm

5 Überblick Flash-Dateisysteme

Eines der ersten Flash-Dateisysteme wurde von Microsoft 1992 FFS2 (vgl. [61]) veröffentlicht, 1994 folgte eNVy, das an der Rice University in den USA entwickelt wurde (vgl. [43]). In den folgenden Jahren wurden weitere Dateisysteme mit unterschiedlichen Stärken und Schwächen entwickelt. In den folgenden Unterabschnitten werden einige ausgewählte Dateisysteme vorgestellt, die eine zwischenzeitlich gewisse Verbreitung haben oder sehr aktuell sind und mit neuen Merkmalen aufwarten.

5.1 Journalling Flash File System (JFFS, 1999)

Bis JFFS vorgestellt und in den Linux-Kernel 2.3 übernommen wurde, war die übliche Vorgehensweise für Flashspeicher, dass eine Flash Translation Layer (FTL) ein Blockzugriffs-Interface bereit stellt. Auf dieser Schnittstelle wurde dann ein gewöhnliches Dateisystem, wie z.B. ext2, eingesetzt. JFFS wurde von Axis Communications AB in Schweden entworfen, um diese ineffiziente Vorgehensweise zu reduzieren, die durch die (damals) rudimentären FTL und Dateisysteme, welche für Festplatten konzipiert wurden, entstehen (vgl. [62], Kap. 1.2). JFFS arbeitet nur mit NOR-Flash zusammen, da NAND zur Zeit der Entwicklung noch keine Rolle gespielt hat, und wurde unter der GPL-Lizenz freigegeben.

Der Aufbau der Schichten ist in Abbildung 25 dargestellt. MTD steht dabei für Memory Technology Device und ist eine Zwischenschicht in Linux, um abstrakt auf Flashspeicher zugreifen zu können. Die drei wichtigsten Funktionen sind das Lesen, Schreiben und Löschen von physikalischen Löschblöcken. Über die MTD-Schicht kann auf verschiedene Flashspeicherarten einheitlich zugegriffen werden. Sie selbst führt kein Wear Leveling, kein Bad Block Management und keine Adressübersetzung durch. Weiterführende Informationen dazu finden sich auf der MTD-Internetpräsenz (vgl. [63]).

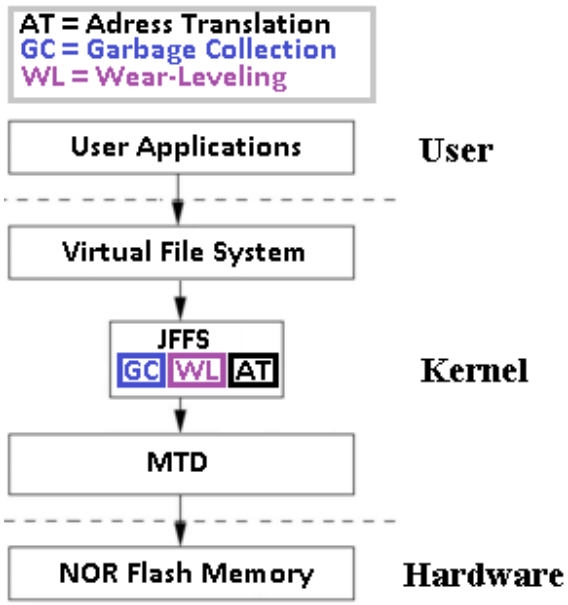


Abbildung 25: JFFS1-Schichtenmodell

JFFS ist ein rein log-basiertes Dateisystem, das direkt (ohne Controller) auf den Flashspeicher zugreift. Es kennt dazu intern eine Struktur namens *jffs_raw_inode*. Jede dieser sogenannten *nodes* davon gehört immer genau zu einer Inode und enthält im Header die aktuellen Metadaten und Identifikationsnummer dieser Inode sowie eine Versionsnummer, die für alle zu einer Inode gehörenden *nodes* eindeutig ist. So kann immer die *node* identifiziert werden, die die aktuellsten Metadaten enthält. Neben dem Header können auch Daten in den *nodes* gespeichert werden, die dann zusätzlich Informationen zu dem Offset enthalten, an dem diese Daten in einer Datei erscheinen sollen. Werden diese Daten oder Metadaten verändert oder überschrieben, wird eine neue *node* mit den aktualisierten Daten und einer höheren Versionsnummer angelegt. Die bisherige *node* gilt dann als veraltet, wird jedoch nicht direkt verändert.

Mountvorgang

Wenn der Flashspeicher gemountet wird (z.B. beim Systemstart), wird jede einzelne *node* gelesen und daraus die vollständige Verzeichnishierarchie und eine Karte zu jeder Inode und den Speicherstellen der Daten im RAM gehalten und dort während der gesamten Systemlaufzeit gespeichert. Dadurch steigt die Mountzeit linear mit der Größe des Flashspeichers an. Dies war 1999 meist noch problemlos möglich, da ein Flashspeicher damals nur einige Megabyte groß war. Heutzutage sind mehrere hundert Gigabyte möglich. Die Wartezeit, bis alle *nodes* beim Systemstart gelesen sind, ist für die

meisten Einsatzgebiete inakzeptabel. Bereits ein Gigabyte Speichergröße kann zu mehreren Minuten Wartezeit führen. (vgl. Kap. 5.6.1, die Mountzeit ist mit JFFS2 vergleichbar).

Garbage Collection

Abbildung 26 stellt einen gesamten Flashspeicher dar, der hier aus sechs Blöcken besteht. Neue und veränderte Daten werden in JFFS immer an das Ende des Logs geschrieben (tail). Wenn neuer Platz benötigt wird, wird immer der Flashblock am Anfang des Logs gelöscht (head). Davor werden gültige *nodes* des zu löschenden Blocks ans Ende geschrieben (Abb. 26 (b) und (c)).

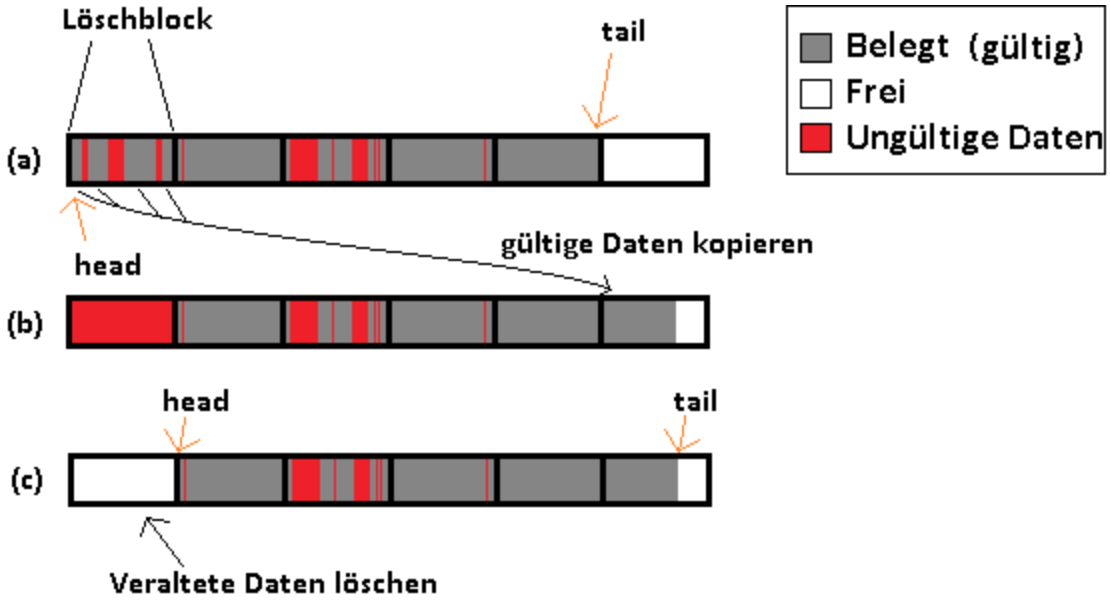


Abbildung 26: Garbage Collection in JFFS

Großer Nachteil bei dieser Vorgehensweise ist, dass dadurch häufig sehr viele gültige Daten kopiert werden müssen, da immer der Flashblock am *head* als *victim block* gewählt wird, gleichgültig ob sich darin wenige oder viele gültige *nodes* befinden. Je voller das Dateisystem ist und je mehr statische Daten es beinhaltet, desto mehr nimmt dieses Problem zu. Dadurch werden viele I/O-Operationen notwendig und die Flashblöcke werden häufig gelöscht und neu beschrieben, wodurch sich die Lebensdauer des Speichers schnell verringert.

Wear Leveling

Es ist kein eigener Wear-Leveling-Algorithmus implementiert, da er unnötig ist. Durch die Vorgehensweise des Garbage Collectors werden alle Blöcke perfekt gleichmäßig abgenutzt.

Da JFFS1 nur mit NOR-Flash arbeitet, ist weder ein Error-Correction-Mechanismus noch Bad-Block-Management vorgesehen (vgl. Kapitel 2.3).

Kritikpunkte sind, dass *hard links* nicht möglich sind (vgl. [62], Abschnitt 2.1, Fußnote 1) und die Daten vom JFFS-Dateisystem nicht komprimiert werden können. Gerade auf den damaligen, sehr kleinen Flashspeichern wurde das von vielen Nutzern gewünscht (vgl. [62], Abschnitt 2.5).

Viele dieser Nachteile wurden in JFFS2 gelöst.

5.2 Journalling Flash File System Version 2 (JFFS2, 2001)

JFFS2 wurde von Red Hat von Grund auf neu geschrieben und verbessert. Es ist ebenfalls unter der GPL veröffentlicht und konnte erstmals ab dem Linux-Kernel 2.4 eingesetzt werden (vgl. [62]). Sowohl NOR als auch NAND-Flashspeicher wird unterstützt. Es wird hauptsächlich in eingebetteten Systemen eingesetzt und ist beispielsweise die Standard-Einstellung in OpenWrt (vgl. [64]), einem Linux-Betriebssystem für kabellose Router.

Der Aufbau ist in Abbildung 27 dargestellt.

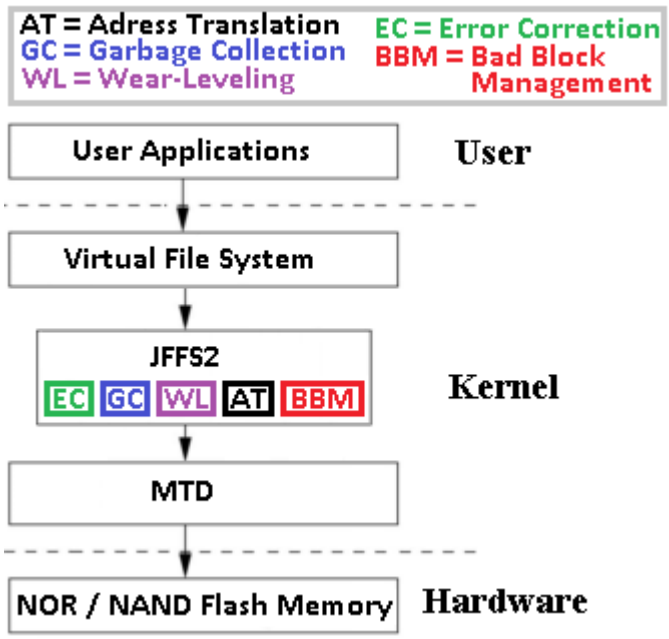


Abbildung 27: JFFS2-Schichtenmodell

Jede *node* in JFFS2 hat im Header zusätzliche Felder: eine Bitmaske, die Größe der *node*, den Header-CRC-Code sowie den *node*-Typ. Die Bitmaske legt fest, wie der Betriebssystem-Kernel mit einer *node* umgehen soll, die für die eingesetzte JFFS-Implementierung unbekannt ist. Vier verschiedene Reaktionen sind möglich, wenn eine solche node gefunden wird: entweder darf das Dateisystem dann (1) gar nicht oder (2) nur als read-only gemountet werden. Oder es kann normal gemountet werden, muss dann aber die unbekannte *node* bei einem Garbage-Collection-Lauf entweder (3) löschen oder (4) wie eine gültige *node* kopieren.

Es gibt mehrere *node*-Typen in JFFS2, genannt Inode, dirent und cleanmarker. Der Inode-Typ entspricht der Standard-node aus JFFS1 und kann nun auch komprimiert werden, z.B. mit zlib. Eine Cleanmarker-node wird in einen gerade gelöschten Block geschrieben, um damit anzuzeigen, dass dieser noch unbenutzt ist. Das ist für den Fall relevant, wenn während eines Block-Löschvorgangs der Strom ausfällt. Ein Block kann dann in einen Zustand geraten, in dem er beim Systemstart als leer identifiziert wird, jedoch einige Bits sich als instabil erweisen. Dann können Daten verloren gehen, die neu in diesen Block geschrieben werden. Die dirent-node dient zum Speichern von Verzeichniseinträgen.

Eine Löschblock-Verwaltung wurde eingeführt, die jeden Block als unabhängige Einheit behandelt. Jeder Block gehört genau zu einer von drei verkettete Listen, der clean_list, dirty_list oder free_list. Blöcke in denen sich ausschließlich gültige *nodes* befinden, gehören zur clean_list. Ist auch nur eine *node* ungültig, wird der Block der dirty_list zugeordnet. In der free_list werden alle leeren Blöcke vermerkt.

Mountvorgang und Hauptspeicherbedarf

Genau wie der Vorgänger wird bei Systemstart erneut jede *node* gelesen und daraus die Verzeichnishierarchie und die Inode-Informationen inklusive zugehörigen Datenblöcken im RAM aufgebaut. Neu ist hingegen, dass nur die Informationen durchgehend im RAM gehalten werden, die nicht sehr schnell rekonstruiert werden können. Andere können bei Bedarf ihren Hauptspeicher freigeben. So sind beispielsweise nicht alle Daten-*node*-Adressen jeder Datei immer im RAM, sondern nur die zugehörige Inode, die auf die Daten-*nodes* verlinkt.

Jedoch skaliert der Hauptspeicherbedarf und die Mountzeit weiterhin linear mit der Anzahl der *nodes*, die in JFFS2 standardmäßig bis zu 4 KB groß sind. Für größere Speicher wurden Veränderungen vorgeschlagen, mit denen die *nodes* größer werden können und damit weniger Hauptspeicher benötigen, jedoch weiterhin linear skalieren (vgl. [65]).

Um die Mountzeit zu reduzieren, wurde ab Linux-Kernel 2.6.15 sogenannte Erase Block Summaries eingeführt. Am Ende jedes Löschblocks wird in einigen *nodes* eine Zusammenfassung aller *nodes* dieses Blocks gespeichert. Dadurch muss nur noch dann jede einzelne *node* des Blocks gelesen werden, wenn diese „Erase Block Summary“-Blöcke fehlen. Dadurch wird die Mountzeit um den Faktor 4-5 beschleunigt, auf Kosten eines Speicherplatz-Overheads von ca. 5% (vgl. [66]). Dennoch haben Performancetests gezeigt, dass das Problem dadurch nicht gelöst werden kann. Die Mountzeit beträgt bei einem zu 75% beschriebenen 256 MB Flashspeicher ca. drei Minuten (vgl. [24], S. 9).

Später wurde zusätzlich eine „Centralized Summary“ eingebaut (vgl. [67]). Durch diese werden alle Dateisystem-Informationen, die zum Mounten notwendig sind, ohne die *nodes* zu scannen, bei jedem Unmount-Vorgang auf den Flashspeicher geschrie-

ben. Der nächste Mountvorgang ist danach deutlich schneller, kann aber immer noch mehrere Sekunden benötigen (vgl. [68], S. 406).

Garbage Collection

Die Garbage Collection wird gestartet, wenn weniger als 5 freie Blöcke verfügbar sind (vgl. [62], Abschnitt 4.2). Ursprünglich und anders als bei JFFS1 wird genau ein *victim block* zu 99% Wahrscheinlichkeit aus einer der Blöcke der dirty_list und zu 1% ein Block der clean_list ausgewählt. Damit wird sichergestellt, dass auch Daten statischer Blöcke gelegentlich verschoben werden. In späteren Versionen wurden noch weitere Listen implementiert, z.B. very_dirty_list (viele ungültige *nodes* vorhanden) und erasable_list (alle *nodes* ungültig) und die Wahrscheinlichkeiten zur Auswahl des *victim blocks* angepasst. Aktuell wird zu ca. 40% ein Block aus der erasable_list gewählt, zu ca. 60% aus der dirty_list und very_dirty_list und weiterhin zu 1% aus der clean_list (vgl. [66]).

Die noch gültigen Daten, die bei der Säuberung kopiert werden müssen, werden in einen freien Zielblock kopiert.

Wear Leveling

Es ist zusätzlich zur Garbage-Collection-Vorgehensweise kein weiterer Wear-Leveling-Mechanismus implementiert. Das dadurch erreichte dynamische und statische Wear-Leveling wird immer Partitionsweise durchgeführt. Gibt es weitere Partitionen mit beispielsweise rein statischen Daten, werden deren Blöcke nicht mit einbezogen (vgl. [24], S. 26).

Nachdem JFFS2 auch mit NAND-Speicher genutzt werden kann, muss es mit defekten Blöcken umgehen können. Wenn ein Block gelöscht werden soll und der Löschversuch drei Mal fehl schlägt, wird er im OOB-Bereich entsprechend markiert, in die bad_block_list aufgenommen und nicht weiter verwendet (vgl. [69]).

Insgesamt wurden die größten Kritikpunkte an JFFS1 in der zweiten Version behoben. Kompression ist nun möglich und auch der Hauptspeicherbedarf wurde reduziert, steigt jedoch immer noch linear mit der Speicherplatzgröße. Auch das Mounten des

Dateisystems dauert ohne Centralized Summary weiterhin sehr lange und daher eignet sich bei den heutigen Speichergrößen nur noch bedingt für den Einsatz.

5.3 Yet Another Flash File System v1 und v2 (YAFFS, 2002)

Ziel von YAFFS[31] war es, ein von Grund auf neues Flash-Dateisystem zu entwerfen, das für NAND-Flashspeicher optimiert ist (vgl. [70]). Es ist modular entwickelt worden und hat die Betriebssystem-spezifischen Programmcodeabschnitte von den Dateisystemcodeabschnitten getrennt, um einfacher portierbar zu sein. Die ursprüngliche Version 1 wurde 2002 veröffentlicht und konnte mit NAND-Flash umgehen, dessen Seitengröße 512 Byte betrug. Mittlerweile werden Seiten bis 1 KB Größe unterstützt. Die 2005 veröffentlichte Version 2 (YAFFS2) ist für größere Seiten ab 1KB ausgelegt, kann jedoch auch kleinere Seitengrößen nutzen.

YAFFS wird bereits in vielen Geräten für sehr unterschiedliche Zwecke eingesetzt und ist verfügbar unter Linux (als Patch, unter GPL), WindowsCE, eCos[32], pSOS[33] und andere. Das Schichtenmodell ist in Abbildung gezeigt.

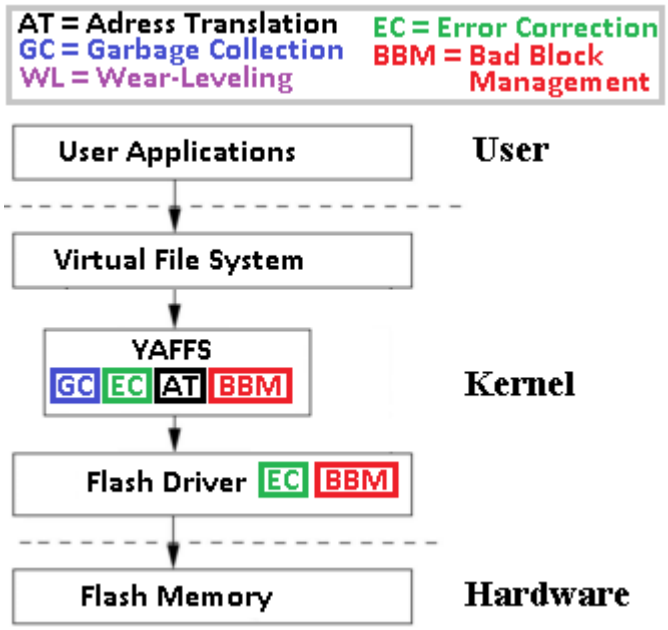


Abbildung 28: Schichtenmodell YAFFS

[31] Im weitere beziehen sich alle Aussagen auf alle Versionen von YAFFS, außer es ist explizit YAFFS1 oder YAFFS2 angegeben

[32] eCos steht für embedded Configurable operating system und ist ein freies Open-Source-Echtzeit-Betriebssystem. Weitere Informationen sind unter http://ecos.sourceware.org/ verfügbar

[33] pSOS bedeutet „Portable Software On Silicon“ und ist ein proprietäres Echtzeit-Betriebssystem

Es ist ein log-basiertes Dateisystem und arbeitet intern mit *chunks* (dt. Stücke), die in der Regel[34] genau so groß sind, wie eine physikalische Seite im Flashspeicher. Zwei Arten von *chunks* werden unterschieden: es gibt *data chunks* und Objekt-Header. Erstere speichern Inhalte von Dateien, in zweiteren werden die Eigenschaften von Objekten (z.B. Verzeichnis, Datei, hard link, soft link, Spezial-Objekt, ...) festgelegt. Weiterhin werden zu jedem *chunk* bis zu fünf *tags* im „Out-of-Band"-Bereich gespeichert. In beiden YAFFS-Versionen gibt es die ObjectID, ChunkID und den ByteCount. In YAFFS2 wurden eine Löschmarkierung (*deletion marker*) und eine Seriennummer (*Serial Number*) nicht mehr weiter geführt.

TAG	***ObjectID***	***ChunkID***	***ByteCount***	***Deletion Marker***	***Serial Number***
Beschreibung	Zu welchem Objekt der *chunk* gehört	An welchen Offset dieser *chunk* gehört	Anzahl an Bytes in diesem *chunk*	Zeigt an, dass dieser *chunk* nicht mehr benötigt wird (nur YAFFS1)	Um *chunks* mit gleicher ObjectID und ChunkID unterschreiben zu können (nur YAFFS1)

Weitere Informationen, wie beispielsweise Verwaltungstabellen, werden in YAFFS nicht auf dem Flashspeicher abgelegt. Aus allen *chunks* auf dem Speicher kann jederzeit der Dateisystemzustand wieder rekonstruiert werden.

Mountvorgang und Hauptspeicherbedarf

Wenn genügend Platz vorhanden war, wurde beim Unmounten ein Checkpoint auf den Flashspeicher geschrieben (seit 2006), in dem die dynamischen Verwaltungstabellen zwischengespeichert werden [70], [71]. Dazu werden 10 freie Blöcke benötigt. Der Checkpoint wird beim nächsten Mountvorgang gelesen, das Dateisystem ist damit in weniger als einer halben Sekunde verfügbar (vgl. [24], S. 9 und [71]). Sobald Daten im Dateisystem verändert werden, wird ein Checkpoint ungültig und gelöscht.

Wird keiner oder ein ungültiger Checkpoint gefunden, müssen alle *tags* der *chunks* des Dateisystems gescannt werden. Durch die daraus gewonnenen Informationen können die Verwaltungstabellen im RAM aufgebaut werden. Seit 2011 gibt es in jedem Block einen *chunk*, das alle *tags* der anderen *chunks* dieses Blocks enthält und viel schneller gelesen werden kann als die einzelnen *tags*. Wenn das *chunk* vorhanden ist,

[34] Wenn beispielsweise zwei Flashchips parallel angesprochen werden können, kann ein chunk auch doppelt so groß wie eine Flashseite definiert werden, um die Perfomance ähnlich wie ein RAID 0 zu steigern

wird es gelesen statt den gesamten Block zu scannen. Dadurch wird die benötigte Zeit zum Mounten ebenfalls stark verringert, falls kein Checkpoint vorhanden ist.

Beim Hauptspeicherbedarf liegen YAFFS2 und JFFS2 etwa gleich auf. Der Bedarf für YAFFS skaliert linear mit der Anzahl *chunks* (vgl. [70], S. 18).

Garbage Collection

Für den Garbage Collector werden immer drei Blöcke frei gehalten. Es wird jeweils nur ein *victim block* ausgewählt, und zwar immer derjenige mit den meisten ungültigen Seiten (Greedy-Algorithmus, vgl. [71]). YAFFS kann auch so konfiguriert werden, dass der Garbage Collector einen eigenen Thread erhält und im Hintergrund arbeitet, wenn das System sich im Leerlauf befindet.

Wear Leveling

Es gibt in YAFFS kein eigenes Wear Leveling. Es davon ausgegangen, dass durch die log-basierte Dateisystemstruktur die Lösch- und Schreibvorgänge gut genug verteilt werden (vgl. [70], S. 20). Da jedoch der Greedy Garbage Collector niemals einen Block säubert, der rein aus gültigen Seiten besteht, ist diese Annahme nicht gerechtfertigt. So zeigt sich wie erwartet in einem Wear-Leveling-Benchmark, dass statische Blöcke nicht einbezogen werden (vgl. [24], S. 25). Hier besteht Verbesserungspotential: es sollte unbedingt auch statisches Wear Leveling integriert werden, damit die Lebensdauer des gesamten Flashspeichers durch gleichmäßige Abnutzung verlängert wird.

Falls mehrere Partitionen auf einem Flashspeicher angelegt sind, findet das Wear Leveling nur in der eigenen Partition statt.

Error Correction und Bad Block Management

Error Correction kann entweder durch YAFFS1 oder durch andere Mechanismen zur Verfügung gestellt werden, wie z.B. Flashtreiber oder die Flashhardware. YAFFS2 erwartet, dass der Flashtreiber die Fehlerkorrektur übernimmt.

Ein Flashblock wird von YAFFS als „bad block“ markiert und nicht weiter benutzt, wenn ein Schreib- oder Lesezugriff auf diesen Block fehlschlägt oder drei ECC-Fehler aufgetreten sind (vgl. [70], Abschnitt 11).

5.4 Unsorted Block Images File System (UBIFS, 2008)

Ursprünglich als JFFS3 geplant und später umbenannt, wurde das Unsorted Block Images File System (UBIFS) seit 2007 von Nokia-Mitarbeitern und der Universität von Szeged (Ungarn) entwickelt (vgl. [72]). Es ist seit dem Linux-Kernel 2.6.27 (2008) in einer stabilen Version verfügbar und ebenfalls unter GPL veröffentlicht. Ziel der Entwicklung war es, einige Schwächen von JFFS2 zu beseitigen, z.B. die Probleme der Skalierbarkeit bei immer größer werdendem Flashspeichern (vgl. [54], Folie 9). Es werden sowohl NOR- als auch NAND-Flash unterstützt.

Das Dateisystem UBIFS setzt im Gegensatz zu JFFS oder YAFFS selbst auf eine weitere, zusätzliche Zwischenschicht namens UBI auf (vgl. Abbildung 29 und [73]), in der UBI-Partitionen erstellt werden können, die dann mit dem eigentlichen Dateisystem UBIFS formatiert werden. Die Hauptaufgabe von UBI ist es, einen Adressübersetzungs-Mechanismus für Blöcke bereitzustellen und somit ein Mapping von logischen Löschblöcken[35] zu physikalischen Löschblöcken durchzuführen. Auch Error Correction für die verwendeten Meta-Daten, Wear Leveling und Bad Block Management wird auf dieser Schicht eingesetzt. I/O-Fehler werden von UBI transparent für die darüber liegenden Schichten behandelt. Durch diese Funktionen vereinfacht sich das Dateisystem entsprechend, da es diese nicht mehr selbst implementieren muss.

UBI

Durch UBI werden wieder drei Operationen angeboten: Lesen, Schreiben und Löschen eines Blocks. Im Gegensatz zur MTD-Schicht werden bei UBI logische Blöcke angesprochen, nicht physikalische. Dadurch sind die zusätzlichen UBI-Funktionen (Wear Leveling, Bad Block Management) möglich.

[35] Von engl. Eraseblock, gemeint ist die kleinste physikalische Einheit, die im Flashspeicher gelöscht werden kann. Im Weiteren wird ein Löschblock einfach als Block bezeichnet

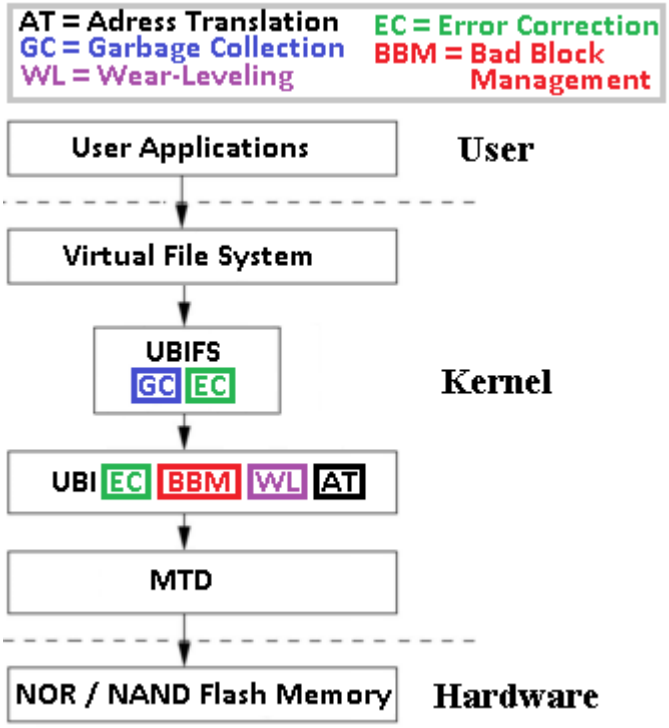


Abbildung 29: UBIFS-Schichtenmodell

Jeder Block in UBI hat zwei Header, die jeweils 64 Byte groß sind. In den ersten wird ein Löschzähler geschrieben, der vermerkt wie oft dieser Block schon gelöscht wurde. Der zweite Header beinhaltet die Information, zu welcher Partition und welchem logischen Block dieser physikalische Block im Moment gehört. Im Falle von NOR-Flash fallen genau diese 128 Byte, die beide Header zusammen benötigen, als Overhead in jedem Block an. Im Falle von NAND-Flash wird eine Seite pro Header verwendet[36], daher haben die logischen Blöcke in UBI immer zwei Seiten weniger als die physikalischen.

Der „Out-of-Band"-Bereich von NAND-Seiten wird von UBI nicht verwendet, da es diesen bei NOR-Flash nicht gibt und UBI mit beiden Flasharten kompatibel sein will.

Mounttime

Beim Mounten müssen die Header jedes Blocks gelesen werden, um das Mapping von physikalischen zu logischen Blöcken im RAM aufzubauen. Das ist deutlich schneller, als jede einzelne node (JFFS) oder jedes chunk (YAFFS) lesen zu müssen

[36] Für den Ausnahmefall, dass der NAND-Flashspeicher es erlaubt, sogenannte „Subpages" zu schreiben, können beide Header in die selbe Seite geschrieben werden

und skaliert hier mit der Anzahl der Blöcke des Flashspeichers, allerdings immer noch linear. Zumindest fast linear, da davon auszugehen ist, dass zukünftige Flashspeicher größere Blöcke verwenden werden.

Es gibt einen Mechanismus, der wie die Checkpoints in YAFFS funktioniert und in UBI als Fastmap bezeichnet wird. Es werden die Zustands-Daten der Blöcke (Löschzähler, Mapping, etc.) zu bestimmten Zeitpunkten, z.B. beim Unmounten der Partition, in dafür vorgesehene Blöcke gespeichert. Beim nächsten Mountvorgang werden diese gelesen. Damit steht die Partition in weniger als einer Sekunde zur Verfügung. Fastmap ist standardmäßig deaktiviert, aber wird bei sehr großen Speichern in Zukunft helfen, dass UBI noch sinnvoll einsetzbar ist.

Wear Leveling

Wenn eine neuer Block benötigt wird, wird immer der freie Block mit dem niedrigsten Löschzähler als nächstes verwendet.

Auch statisches Wear Leveling ist vorhanden. Der Löschzähler eines Blocks wird jedes mal überprüft, wenn ein Block gelöscht wird. Wurde der gerade gelöschte Block mehr als 5000 mal öfter gelöscht (vgl. [74], S. 19), als der Block mit der niedrigsten Löschzahl, wird der Inhalt des Letzteren in den alten Block kopiert, da dieser sehr wahrscheinlich nur Daten enthält, die sich selten ändern.

Gibt es mehrere UBIFS-Partitionen, werden Blöcke auch partitionsübergreifend in das Wear Leveling mit einbezogen.

Error Correction und Bad Block Management

Bei NAND-Speicher werden ca. 2% aller Blöcke als Ersatz für defekte Blöcke reserviert (vgl. [73], Nr. 16). Dieser Wert kann bei Bedarf konfiguriert werden. Ein Block gilt sofort als defekt, wenn das Löschen dieses Blocks fehlschlägt. Sollte eine Lese- oder Schreiboperation auf einen Block fehlschlagen, wird dieser Block für Fehlertests[37] markiert und alle gültigen Daten in diesem Block werden in einen anderen Block kopiert. Die Fehlertests löschen den Block, schreiben und lesen danach mehrere verschiedene Testdaten und markieren ihn nur dann als Bad Block, wenn es bei den Fehlertests erneut zu Problemen kommt.

UBI Header sind über CRC-Checksummen geschützt.

[37] In der UBI-Dokumentation werden diese als „torture tests“ bezeichnet

UBIFS

Um eine bessere Skalierbarkeit der Mountzeit als JFFS2 und YAFFS2 zu erreichen, speichert UBIFS im Gegensatz zu diesen den Index des Dateisystems auch auf dem Flashspeicher anstatt nur im RAM, wo er dann bei jedem Mountvorgang erneut aufgebaut werden müsste.

Der Index ist in Form eines B^+-Baums[38] realisiert, standardmäßig ist der Verzweigungsgrad auf 8 eingestellt (vgl. Abbildung 30, dort mit Verzweigungsgrad 3 dargestellt). Nur in den Datenblättern werden Inhalte wie Inodes, Daten-Offsets und Verzeichniseinträge gespeichert. Wegen des Ablegens des Indexes auf dem Flashspeicher muss allerdings ein neues Problem gelöst werden, da bei jeder Datenänderung auch alle damit verlinkten Index-Knoten des Baums aktualisiert werden müssen. Als Beispiel: wenn ein Datenblock einer Datei verändert wird, werden die veränderten Daten in einen beliebigen anderen Datenblock abgelegt, da es auf Flash nicht überschrieben werden kann. Der Verweis auf den Datenblock im Index muss dann ebenso angepasst werden und wird wiederum an eine neuen Speicherstelle geschrieben. Dadurch muss der Verweis darauf wieder geändert werden. Das bedeutet, dass ein Knoten auf jeder Ebene des Baums bis zum Wurzelknoten angepasst werden müsste, wodurch sehr viele Schreibzugriffe stattfinden würden. Dieses Problem wird als wandernder Baum bezeichnet.

Um es zu lösen, wurde ein Journal eingeführt, in dem jede Änderungen des Baums protokolliert wird. Wenn dieses Journal fast voll ist, werden alle Änderungen des Index gleichzeitig geschrieben („commit“) und wieder von vorne begonnen. Im laufenden Betrieb werden die aktuellen Speicherorte immer im RAM gefunden. Sollte das Dateisystem unsauber beendet werden, z.B. durch einen Stromausfall, so kann der aktuelle Index durch den veralteten Index auf dem Flashspeicher wieder berechnet werden, indem alle im Journal eingetragenen Änderungen übernommen werden.

[38] Ein B^+-Baum ist eine Daten- und Indexstruktur, bei der die eigentlichen Daten nur in den Blattknoten gespeichert werden. Alle anderen Knoten erhalten ausschließlich Verweise auf weitere Knoten.

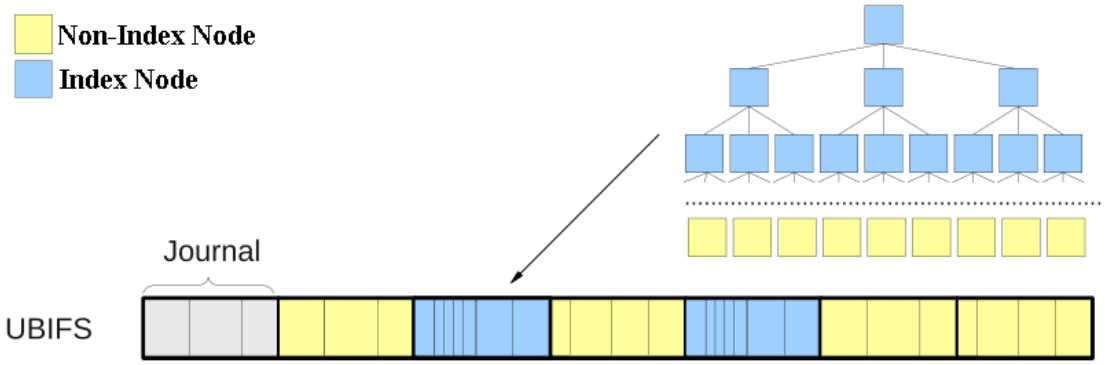


Abbildung 30: Die UBIFS-Indexstruktur (basierend auf [54], Folie 30)

Das Dateisystem teilt den Flashspeicher zur eigenen Verwaltung in sechs Bereiche auf:

1. Superblock Node: diese enthält die Dateisystem-Parameter wie z.B. standardmäßig zu verwendende Kompression, Flash-Eigenschaften, Verzweigungsgrad des Index-Baums. Die Node befindet sich immer im logischen Block 0 und wird normalerweise nie wieder verändert.
2. Master Node: in ihr werden alle Speicherorte von UBIFS-Strukturen abgelegt, die sich nicht an festen Speicherstellen befinden. Die Node wird aus Sicherheitsgründen immer dupliziert und in den logischen Blöcken 1 und 2 gespeichert.
3. Main Area: hier werden alle Daten-, Journal- und Indexblöcke abgelegt.
4. Logbereich: er enthält Nodes, in denen die Journal-Blöcke verlinkt sind, in denen die Dateisystem-Index-Änderungen gespeichert sind
5. Der LPT-Bereich: LPT steht für „LEB Properties Tree“, wobei LEB wiederum für „Logical erase block“ steht, also ein logischer UBI-Block gemeint ist. In dem LPT werden Eigenschaften der logischen Blöcke der Main Area gespeichert. Dazu gehört, wie viel Platz in jedem Block noch frei sind, wie viele ungültige Daten er enthält und ob es sich um einen Indexblock handelt oder nicht. Da sich auch diese Daten extrem oft verändern, wird auch für sie wieder ein Journal mit den Änderungen geführt und nur gelegentlich auf den Flashspeicher zurückgeschrieben. Der aktuelle Wert befindet sich immer im RAM.
6. Orphan Area: hier landen Inodes mit einem Linkcount von 0. Details dazu in [75].

Die Index-Strukturen müssen nicht immer im RAM gehalten werden, UBIFS kann den Speicher für alle Knoten, die aktuell auf nicht benutzte Dateien verweisen, freigeben.

Eine weitere Besonderheit von UBIFS gegenüber anderen Flashdateisystemen ist die Unterstützung von writeback. Dadurch werden Schreibzugriffe auf den Speicher in einem Cache gehalten und im Hintergrund geschrieben. Dadurch kann es zu Effizienzsteigerungen kommen, wenn Schreibzugriffe auf die gleiche Datei zusammengefasst werden können.

Mounttime

Da der Dateisystem-Index auf dem Flashspeicher abgelegt ist und vor jedem Unmount standardmäßig ein commit ausgeführt wird, mountet UBIFS sehr schnell, üblicherweise im Bruchteil einer Sekunde, unabhängig von der Flashgröße oder der Speicherplatzbelegung im Dateisystem (vgl. [75], [76]). Dies bezieht sich allerdings rein auf UBIFS, denn die UBI-Schicht selbst benötigt auch Zeit und muss den Header jedes Blocks lesen, solange Fastmap dort nicht aktiviert ist. Die Mountzeit hängt somit wiederum von der Anzahl der physikalischen Blöcke des Flashspeichers ab. So benötigt das Mounten eines 1 GB NAND-Speichers mit 512KB Blockgröße, also 2000 Blöcken insgesamt, ca. 2.6 Sekunden (vgl. [77], Folie 15). Ohne Fastmap dürfte die benötigte Mountzeit zu hoch sein, etwa für Smartphones, die heutzutage problemlos 16 GB und mehr internen Speicher haben können.

Garbage Collection

Es ist immer ein logischer Block für den Garbage Collector reserviert. Beim Lauf der Garbage Collection wird der Block zur Säuberung gewählt, der die meisten ungültigen Daten enthält („Greedy“, vgl. [73], [75]). Alle noch gültigen Daten werden in den reservierten Block übernommen. Dann wird der gesäuberte Block gelöscht. Danach wird ein weiterer Block gewählt und dessen gültige Daten ebenfalls in den reservierten Block geschrieben und dieser Block ebenso freigegeben. Ist nicht genügend Platz im

reservierten Block vorhanden, wird ein neuer verwendet. Dieser Vorgang wird so lange wiederholt, bis insgesamt ein neuer, freier Block erzeugt wurde.

Hier zeigt sich, dass bei den Daten, die durch den Garbage Collector verschoben werden, keinerlei Aufteilung erfolgt, sondern alle gültigen Daten in den gleichen Block geschrieben werden (vgl. [54], Folie 42). Würde hier eine Trennung, z.B. nach einem der vorgestellten *hot/cold* Trennungsverfahren (vgl. Kap. 4.1.1.3) vorgenommen, dürfte sich die Leistungsfähigkeit der Garbage Collection verbessern lassen.

5.5 Flash-Friendly File System (F2FS, 2012)

F2FS von Samsung wurde Ende 2012 veröffentlicht (vgl. [78]). Es verfolgt einen anderen Ansatz als die bislang vorgestellten Flash-Dateisysteme, da es nur auf Flashspeichern funktioniert, die bereits über eine eingebaute FTL im Hardwarecontroller verfügen und explizit nicht auf „raw flash"-Speicher eingesetzt werden kann. Die Geschwindigkeit und Zuverlässigkeit der Speicher soll durch F2FS verbessert werden. Es zielt auf SD-Karten, eMMC-Karten und SSDs ab und kann aktuell unter Linux (seit Kernel 3.8) eingesetzt werden. Samsung stellt auch Implementationen für die Handys Galaxy S2, S3 und Nexus bereit.

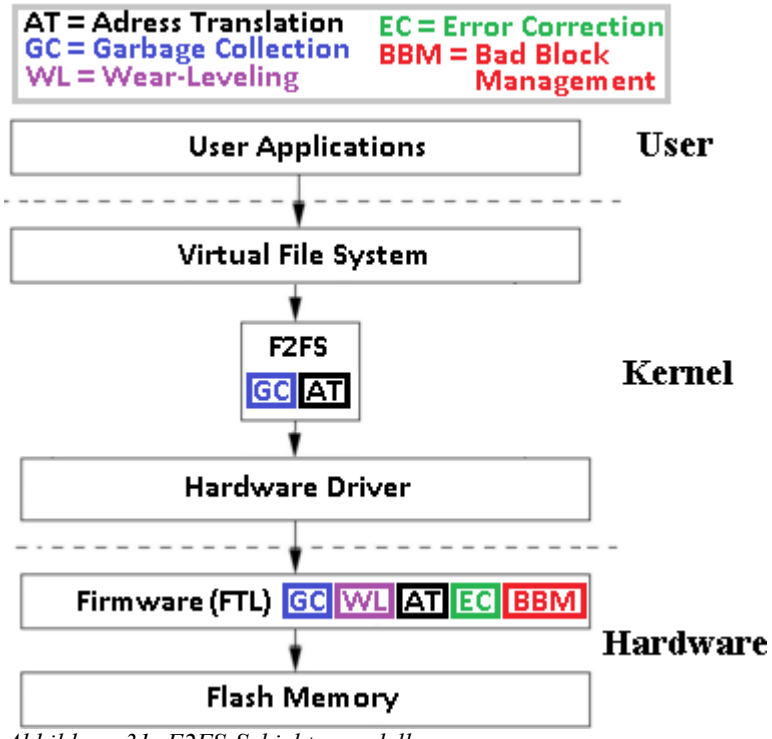


Abbildung 31: F2FS-Schichtenmodell

Der Speicherplatz wird in verschiedene Einheiten unterteilt, wie in Abbildung 32 dargestellt ist.

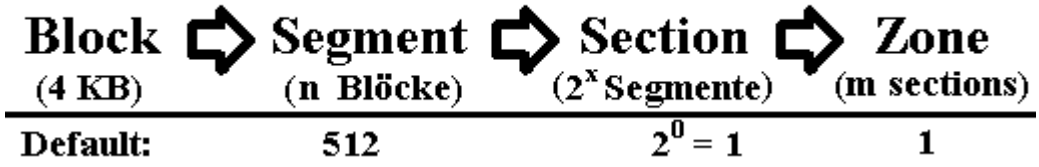


Abbildung 32: Unterteilung des Flashspeichers durch F2FS

Die kleinste Einheit in F2FS ist ein Block, der immer 4 KB groß ist. Die Blockgröße ist ein Vielfaches der Flash-Seitengröße (vgl. [79]). Somit belegt z.B. bei einer Seitengröße des Flashspeichers von 2 KB ein F2FS-Block immer zwei Seiten, bei einer Größe von 1 KB belegt ein F2FS-Block vier Seiten, et cetera.

Blöcke werden zu Segmenten zusammengefasst. Zu jedem Segment gibt es einen Segment Summary Block, in dem gespeichert wird, zu welcher Datei und gegebenenfalls zu welchem Offset jeder Block des Segments gehört.

Segmente können zu Sections zusammengefasst werden, jedoch legt die Standard-Einstellung fest, dass aktuell jede Section nur aus einem Segment besteht. Für Schreibzugriffe werden immer sechs verschiedene Sections offen gehalten, und je nach erwarteter Lebensdauer der zu schreibenden Daten werden sie in einer davon abgelegt. Auf der Section-Ebene arbeitet auch die Garbage Collection von F2FS.

Sections werden zu Zonen zusammengefasst, aber auch hier ist der Default-Wert eins. Das Ziel ist, die sechs offenen Sections in unterschiedlichen Bereichen der Hardware offen zu halten. Der Hintergrund dabei ist, dass viele Flashspeicher intern aus mehreren parallelen Teilen aufgebaut sind, die unabhängig voneinander I/O-Zugriffe verarbeiten können (vgl. [79]). Wenn die Zonen daran angepasst sind, kann durch diese Parallelisierung eine erhöhte I/O-Geschwindigkeit erreicht werden.

Das Dateisystem an sich arbeitet auf zwei großen, unterschiedlichen Strukturen: den Metadaten-Bereich und der „Main Area“ (dt. Hauptbereich). Die folgende Abbildung 33 wird den Aufbau zeigen. Die Main Area enthält alle Inodes und Datenblöcke, wäh-

rend im Metadaten-Bereich Informationen gespeichert werden, die zur Verwaltung der Flashstrukturen notwendig sind.

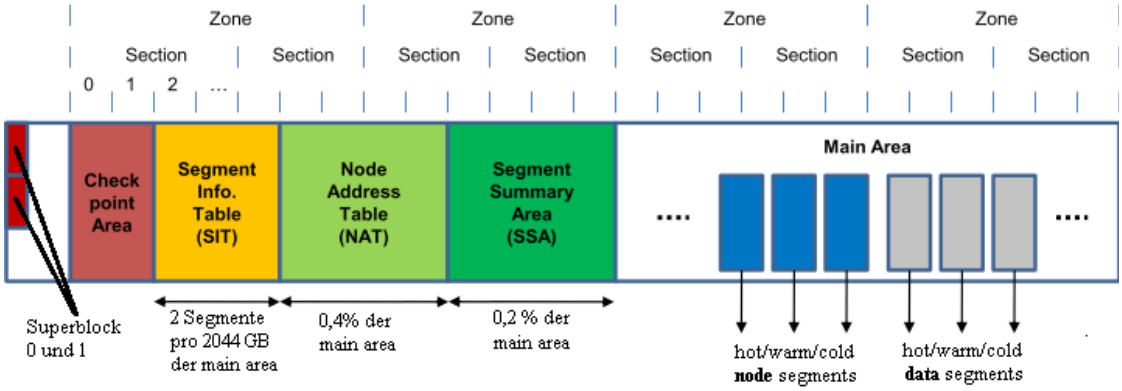


Abbildung 33: F2FS: Logischer Aufbau der Speicherbereiche (vgl. [78], Folie 11)

Zunächst zur Main Area. Hier sind die Inodes und Datenblöcke abgelegt. Die Inodes funktionieren dabei fast genauso wie im UNIX-Dateisystem und seinen Ablegern (z.B. ext3). Jede Inode hat eine Nummer und speichert die Dateiinformationen, darunter auch direkte und (ggf. mehrfach) indirekte Verweise auf die dazugehörigen Datenblöcke. Die Größe einer Datei ist dadurch auf ca. 3,94 Terabyte begrenzt.[39] Auch bei sehr kleinen Dateien werden in der Inode selbst keine Daten abgelegt, somit gibt es zu jeder Datei mindestens eine Inode und einen Datenblock.

Verzeichnisinformationen werden ebenfalls in Blöcken gespeichert. Jedes Verzeichnis kann in einem oder mehreren Blöcken Verweise auf Dateien und Unterverzeichnisse enthalten. Bis zu 214 Verweise pro Block sind möglich (vgl. [80], Folie 26).

Der Metadaten-Bereich setzt sich aus fünf Teilen zusammen (s. Abbildung 33 links, alles außer die Main Area). Der erste davon ist für die Superblöcke reserviert. Dort stehen alle Dateisystem-Informationen die sich niemals ändern, beispielsweise die Größe des Dateisystems, der Segmente, der Sections, der Zonen und auch der anderen Metadaten-Unterbereiche.

In den anderen Bereichen werden dynamische Daten gespeichert. Die Segment Summary Area speichert die bereits zuvor erwähnten Segment Summary Blocks aller

[39] Jeder Verweis zeigt auf einen 4 KB Datenblock. Durch 929 direkte Verweise, zwei indirekte Verweise auf einen weiteren Block mit jeweils 1018 Verweisen, zwei zweifach-indirekten Verweisen und einem dreifach indirekten Verweis kann der Wert berechnet werden:
Es kann auf 929 + 2*1018 + 2*1018*1018 + 1018*1018*1018 Blöcke à 4 KB verwiesen werden

Segmente ab. Dieser Bereich wird nicht wie in einem log-basierten Dateisystem verwaltet, sondern Updates werden hier In-Place durchgeführt. Da auf einer tieferen Schicht ein Controller FTL-Funktionen durchführt, sieht es für das Dateisystem wie ein In-Place-Update aus. Dies kann bei einem Absturz zu ungültigen Daten in diesem Bereich führen, allerdings lassen sich die Informationen des Segment Summary Blocks aus den Daten der Main Area rekonstruieren.

Der Bereich für den Node Adress Table (NAT) dient dazu, das Problem mit den wandernden Bäumen (vgl. Abschnitt 5.4) zu verringern, in dem bei einer Adressänderung eines Blocks auch alle Verweise auf den Block neu geschrieben werden müssen. Verweise in Inodes oder Indexblöcken verweisen nicht auf die Adresse des gesuchten Blocks, sondern auf einen Eintrag im NAT, in dem die gesuchte Adresse verlinkt wird. Änderungen einer Adresse müssen nur in der NAT angepasst werden (vgl. [81], Folie 14). Auch für diese Realisierung setzt F2FS auf die FTL einer tieferen Schicht. Updates in der NAT werden nicht sofort geschrieben, sondern im Hauptspeicher vorgehalten und zu bestimmten Zeitpunkten auf einmal abgespeichert.

Im Segment Info Table wird abgelegt, welche Blöcke der Segmente noch gültig sind Für die Garbage Collection sind diese Daten sehr wichtig. Da sich diese Informationen viel häufiger ändern als diejenigen, die in der Segment Summary Area abgelegt sind, werden sie getrennt davon gespeichert (vgl. [79]). Auch hier werden Änderungen im Hauptspeicher zwischengespeichert und gemeinsam geschrieben.

Zuletzt gibt es noch die „Checkpoint Area“. Diese speichert unter anderem wie viel freier Speicherplatz noch verfügbar ist oder die Adresse der Segmente, in die als nächstes geschrieben werden sollen. Der Checkpoint wird beim Mounten des Dateisystem gelesen. Es gibt immer zwei Checkpoints, von denen einer aktuell ist.

Parallelisierung und Hot/Cold-Trennung

Es werden immer sechs verschiedene Sections für Schreibzugriffe bereit gehalten. Wohin genau ein Schreibzugriff tatsächlich geschrieben wird, hängt von der Art der zu schreibenden Daten ab (vgl. [81], Folie 16 und [80], Folie 30). Es wird jeweils zwischen hot/warm/cold bei Indexblöcken und Datenblöcken unterschieden und je nach Klassifizierung eine der sechs Sections als Ziel ausgewählt.

Garbage Collection

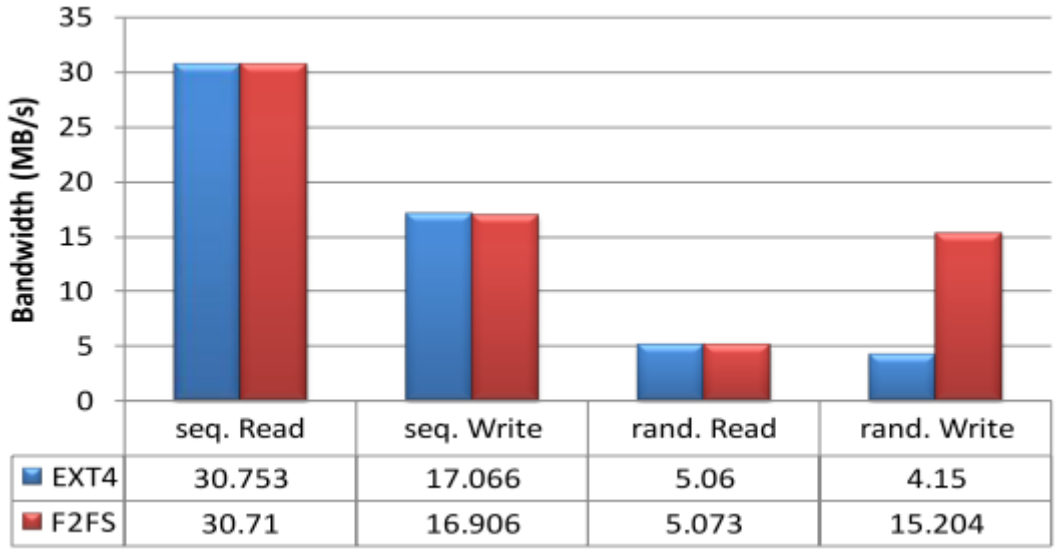
Es werden zwei verschiedene Garbage Collectoren eingesetzt. Wenn für Schreibzugriffe neuer Speicherplatz geschaffen werden muss, wird ein Garbage Collector mit Greedy-Algorithmus eingesetzt (vgl. [78], Folie 13).

Es gibt weiterhin einen Kernel-Prozess, der einen Garbage-Collection-Hintergrund-Job aktiviert, wenn sich das System im Leerlauf befindet (vgl. [80], Folie 33). Dieser nutzt den Cost-Benefit-Algorithmus um freie Sections zu schaffen und teilt die zu kopierenden, noch gültigen Daten wiederum nach hot/cold auf (vgl. [81], Folie 17).

Leistungsvergleich

In Geschwindigkeitstests zeigt sich, dass F2FS insbesondere bei zufälligen Schreibzugriffen gegenüber ext4 starke Verbesserungen zeigt, ansonsten aber zumindest gleichauf liegt. Ein Vergleich mit den anderen Flashdateisystemen wie YAFFS oder UBIFS ist nicht sinnvoll, da diese nur auf Flashspeicher ohne zwischengeschalteten FTL-Controller funktionieren und daher nicht im gleichen Umfeld eingesetzt werden können.

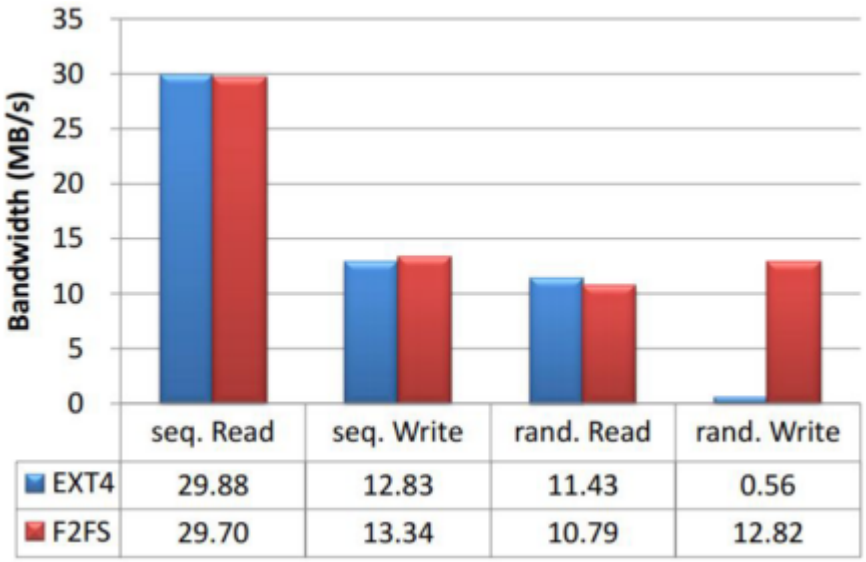
Der in Abbildung 34 gezeigte Tests wurde auf einer eMMC-Karte mit Linux-Kernel 3.3 und iozone (vgl. [82]) durchgeführt.



	seq. Read	seq. Write	rand. Read	rand. Write
EXT4	30.753	17.066	5.06	4.15
F2FS	30.71	16.906	5.073	15.204

Abbildung 34: F2FS Geschwindigkeitsvergleich auf eMMC (vgl. [81], Folie 19)

Auch auf einem Smartphone Galaxy Nexus mit Android 4.0.4 bringt F2FS deutliche Geschwindigkeitssteigerungen bei zufälligen Schreibzugriffen, wie ein iozone-Vergleich in Abbildung 35 zeigt.



	seq. Read	seq. Write	rand. Read	rand. Write
EXT4	29.88	12.83	11.43	0.56
F2FS	29.70	13.34	10.79	12.82

Abbildung 35: F2FS Geschwindigkeitsvergleich auf Android-Smartphone (vgl. [80], S. 38)

5.6 Performance-Vergleiche

In vielen Veröffentlichungen wurden Geschwindigkeitsvergleiche zwischen verschiedenen Flash-Dateisystemen gezeigt. Für die hier vorgestellten Dateisysteme werden beispielhaft einige davon vorgestellt. Da F2FS aufgrund seiner Architektur[40] nicht in direkter Konkurrenz zu den anderen in dieser Arbeit beschriebenen Dateisystemen steht, sind die F2FS-Vergleichstests in Kapitel 5.5 dargestellt.

5.6.1 Mount Time

Durch sie soll festgestellt werden, wie lange nach dem Einschalten eines Geräts gewartet werden muss, bis das Dateisystem zur Verfügung steht. Insbesondere bei Endkundengeräten ist dieser Wert von großem Interesse, da es an diese kaum vermittelbar wäre, wenn eine Digitalkamera, Tablet oder ein Smartphone mehrere Minuten zum Starten benötigt.

[40] F2FS ist als Dateisystem für Flashspeicher mit einem in Hardware integriertem FTL-Controller konzipiert, während JFFS2/YAFFS2 und UBIFS nur auf raw flash devices arbeiten

Ein Vergleich wurde von J. Jung zwischen YAFFS2 und UBIFS durchgeführt (vgl. [77]). Als Testsystem diente ein System on a Chip mit 400 MHz CPU, 512 MB RAM und 2 GB NAND-Flashspeicher mit 512KB Blöcken (=4096 Blöcke insgesamt). Zunächst wurde die Mounttime nach einem vorherigen, sauberen Unmount-Vorgang gemessen, wie in Abbildung 36 gezeigt. Auf der X-Achse ist die verwendete Größe der Partition in MB und die Speicherplatzbelegung des Dateisystems in Prozenten angegeben.

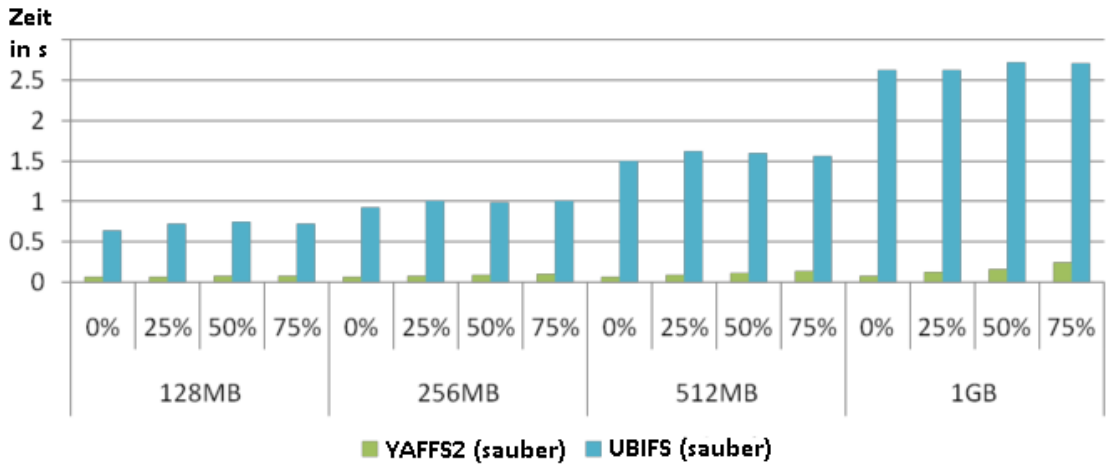


Abbildung 36: Mountime-Vergleich (sauber) zwischen YAFFS2 und UBIFS (vgl. [77], Folie 15)

Hier zeigt sich, dass YAFFS2 nahezu konstant im Bereich von 0,1 Sekunden mountet, während UBIFS mit der Größe der Partition skaliert, dafür unabhängig von der Speicherplatzbelegung des Dateisystems. Das Ergebnis von YAFFS2 ist schnell erklärt: da das System vorher sauber heruntergefahren wurde, hat es einen Checkpoint mit den Dateisysteminformationen auf dem Flashspeicher abgelegt und muss nur diesen beim nächsten Mounten lesen. UBIFS muss, solange fastmap in UBI nicht aktiviert wurde, die Header aller zum Dateisystem gehörenden Blöcke lesen und skaliert daher mit der Partitionsgröße.

Der Vorteil von YAFFS2 gegenüber UBIFS dreht sich jedoch schnell ins Gegenteil, wenn das Dateisystem nicht sauber unmountet wurde, wie in Abbildung 37 gezeigt. Dort wurde mitten in einem Schreibvorgang der Strom abgeschaltet und es konnte da-

her kein Checkpoint in YAFFS2 erstellt werden, auch UBIFS konnte sein Journal mit den Dateisystem-Index-Änderungen nicht zurückschreiben.

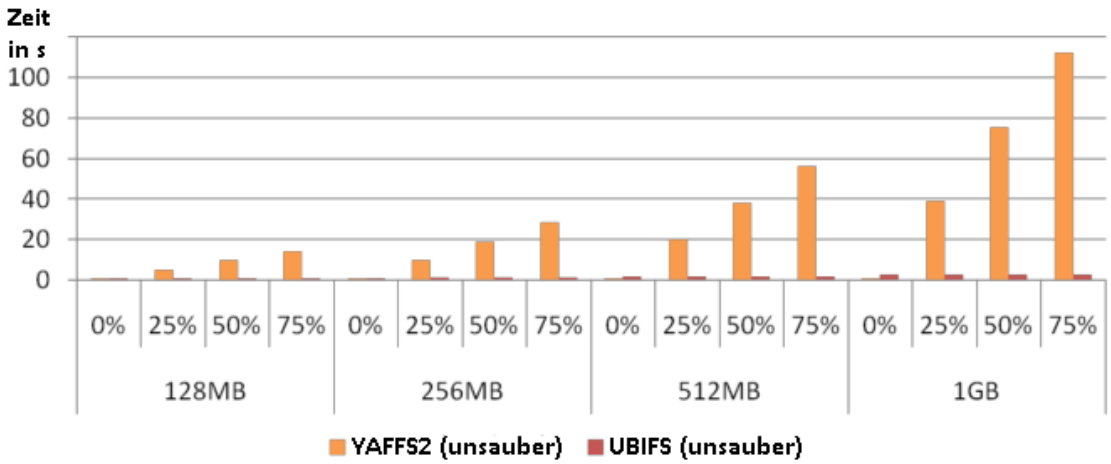


Abbildung 37: Mountime-Vergleich (unsauber) zwischen YAFFS2 und UBIFS (vgl. [77], Folie 16)

Im Ergebnis zeigt sich, dass UBIFS dennoch extrem schnell gemountet werden kann, da es nur die noch vorhandenen Journal-Einträge in den Dateisystem-Index, der auf dem Flashspeicher liegt, nachziehen muss. YAFFS2 hat keinen derartigen Mechanismus und muss daher den „Out of Band"-Bereich jedes *chunks* auf dem gesamten Dateisystem lesen, um die Dateisystem-Strukturen im RAM neu aufzubauen. Daher skaliert die Mounttime bei YAFFS2 mit der Anzahl der *chunks* des Dateisystems. Bereits bei einem nur halb belegten 1 GB-System dauert es über eine Minute, bis das Dateisystem wieder bereit ist.

Ein weiterer Vergleich, diesmal zwischen JFFS2, YAFFS2 und UBIFS, wurde auf der CELF 2009 gezeigt (vgl. [24]). Das Testsystem dazu ein System mit 327 MHz CPU, 256 MB RAM und einem 256 MB NAND-Flashspeicher mit 128 KB Blöcken. Die Dateisystem-Belegung wurde durch eine einzige Datei erreicht. Es wurde die Zeit gemessen, ab dem das Dateisystem als erfolgreich gemountet gemeldet wurde und nachdem ein erster LS-Befehl durchgeführt wurde.

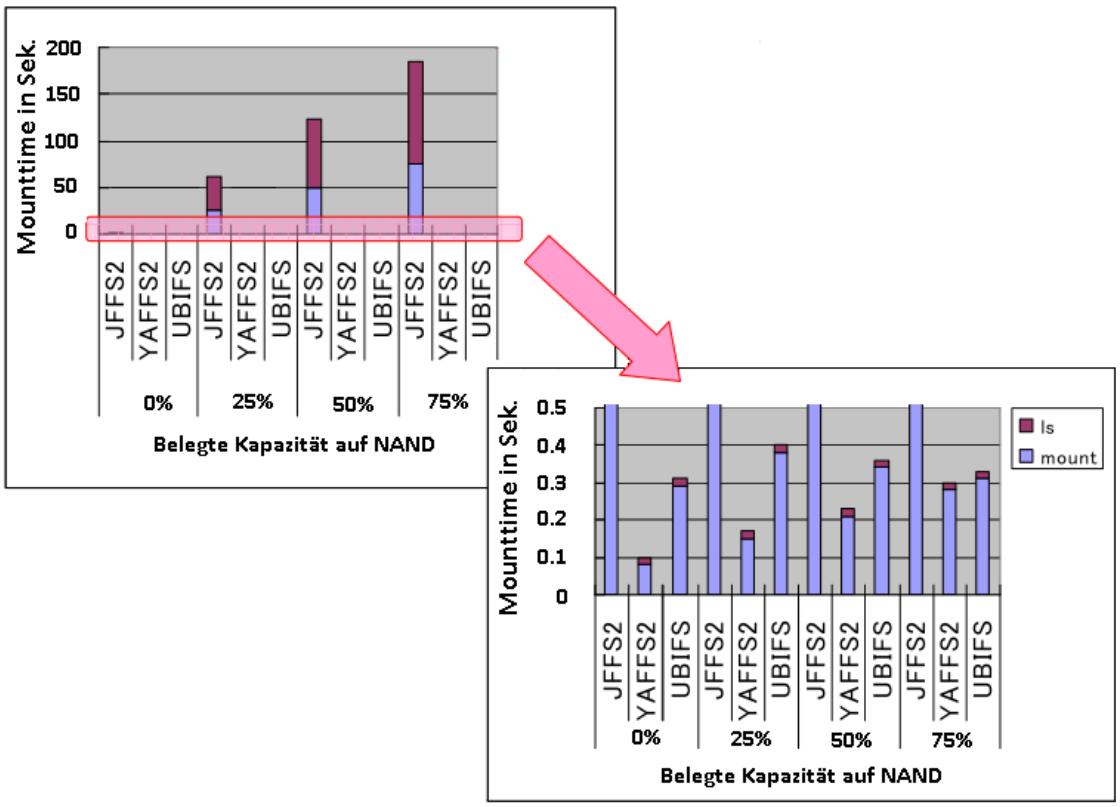


Abbildung 38: Mounttime-Vergleich JFFS2, YAFFS2 und UBIFS (vgl. [24], Folie 9)

Hier zeigt sich, dass JFFS2 ohne „Centralized Summary“ benutzt wurde, da die Zeit zum Mounten abhängig von der Belegung des Dateisystems ist. YAFFS2 hingegen mountet sehr schnell und benutzt daher offensichtlich die Checkpoint-Funktion. Erstaunlich ist, dass durch JFFS2 nach dem erfolgreichen Mounten nochmal etwa gleich viel Zeit benötigt wurde, bis der LS-Befehl ausgeführt war. UBIFS war in diesem Test wie erwartet bei jedem Szenario gleich schnell, da sich die Anzahl der Flashblöcke nicht verändert hat.

Insgesamt lässt sich feststellen, dass die Zeit zum Mounten durch die Checkpoint-Funktionen aller Dateisysteme[41] immer sehr schnell ist, solange das Dateisystem vor dem Abschalten diese schreiben konnte. Wenn im Crash-Fall solche Zusammenfassungen nicht vorhanden waren, ist die Zeit bei JFFS2 und YAFFS2 stark angestiegen, so dass bereits ab Speicherkapazitäten von 1 Gigabyte mehrere Minuten eingerechnet werden mussten. UBIFS funktioniert hier deutlich besser: UBI muss dann nur wenige Daten pro Block lesen und UBIFS speichert in periodischen Abständen seinen

[41] „Centralized Summary“ bei JFFS2, „Checkpoint“ bei YAFFS2, „fastmap“ in der UBI-Schicht

Checkpoint und muss im Falle eines unsauberen Starts nur die Änderungen aus seinem Journal nachvollziehen. Je nach Größe des Speichers kann es damit um den Faktor 100 schneller sein als YAFFS2 und JFFS2.

5.6.2 Hauptspeicherverbrauch

Der vom Dateisystem verbrauchte Hauptspeicher hängt einerseits von den Modul-Programmdaten ab und andererseits von den dynamischen Strukturen, die zur Dateisystem-Verwaltung (z.B. Index, Löschzähler) benötigt werden.

Die Programmdaten der Module belegen üblicherweise nur Hauptspeicher im Bereich 50 KB bis 250 KB, wie in Abbildung 39 gezeigt. Das ist praktisch für jedes System heutzutage keine Hürde mehr.

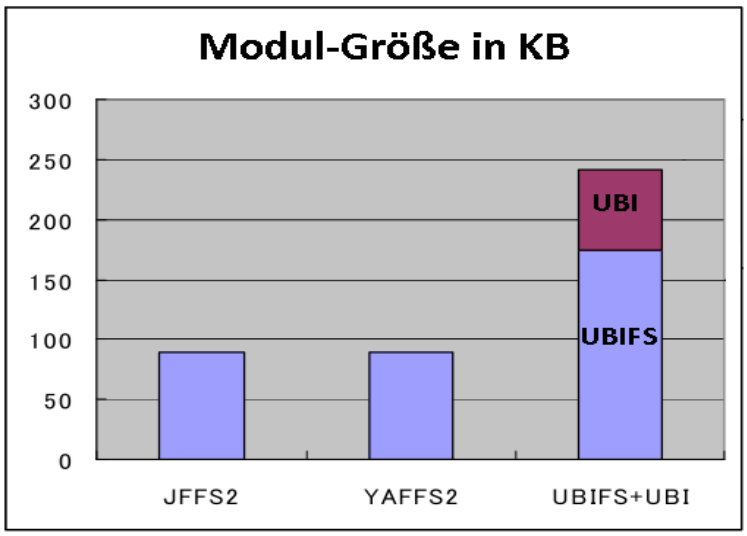


Abbildung 39: Hauptspeicherbelegung der Dateisystemmodule (vgl. [24], Folie 15)

Der Hauptspeicherbedarf im laufenden Betrieb hängt von der Anzahl der Dateien ab, die im Dateisystem existieren. Es wurden verschieden viele Dateien mit jeweils 1 KB Größe angelegt (Abbildung 40 links). Der gleiche Test wurde mit nur einer Datei im Dateisystem durchgeführt, die 0, 1 und 10 MB groß war (Abbildung 40 rechts).

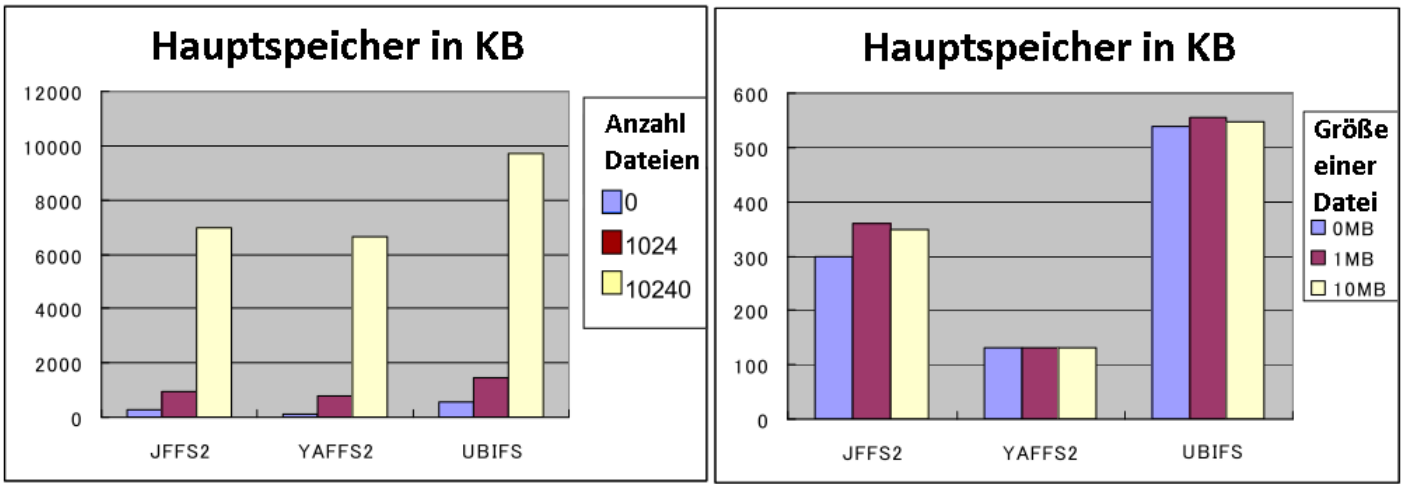


Abbildung 40: Hauptspeicherbelegung mit unterschiedlichen Dateien (vgl. [24], Folie 17f.)

Es zeigt sich, dass der Hauptspeicherbedarf von der Größe der Dateien unabhängig ist, jedoch von der Anzahl der Dateien abhängt. Dies wird damit zusammenhängen, dass jede Datei eine eigene Inode zur Verwaltung benötigt. Wenn das System mehr Hauptspeicher benötigt, können JFFS2 und UBIFS einigen Speicher freigeben.

5.6.3 Schreib- und Lesegeschwindigkeit

Es wurde ein Postmark-Test (vgl. [68], S. 406f.) mit den Einstellungen 1000 Dateien, 10-15000 Bytes/Datei, 10000 Transaktionen[42] durchgeführt. Das Testsystem bestand aus einem 2 GB NAND-Flashspeicher (128 KB pro Block). Das Ergebnis dieses Tests ist in Tabelle 2 zusammengefasst.

	JFFS2	YAFFS2	UBIFS
Zeit gesamt (s)	590	552	177
Lesedurchsatz (KB/s)	410	438	1340
Schreibdurchsatz (KB/s)	428	458	1390

Tabelle 2: I/O-Durchsatzvergleich JFFS2, YAFFS2 und UBIFS (vgl. [68], Table 6)

JFFS2 und YAFFS2 sind ungefähr gleich schnell. UBIFS zeigt einen sehr hohe Vorsprung. Das müsste am writeback-Cache liegen, der bei den gewählten Testeinstellungen einen großen Vorteil hat, da bei vielen Transaktionen Dateien schnell erstellt und wieder gelöscht werden. Diese müssen nicht alle auf den Flashspeicher geschrieben werden, wenn sie rechtzeitig wieder gelöscht wurden, bevor der Cache geleert wird. Wenn sehr große Dateien geschrieben werden oder geschriebene Daten erst mit einiger Zeitverzögerung wieder gelöscht werden, dürfte der Vorteil sich stark verringern.

[42] Eine Transaktion führt in diesem Fall paarweise die Operationen (a) Dateien erstellen/Datei löschen oder (b) Datei lesen/an Datei anhängen aus

In einem ähnlich angelegten Tests, das auf einem 1 GB NAND-Flashspeicher (wiederum 128 KB pro Block) durchgeführt wurde, zeigt sich ein engeres Ergebnis, wie in Abbildung 41 gezeigt.

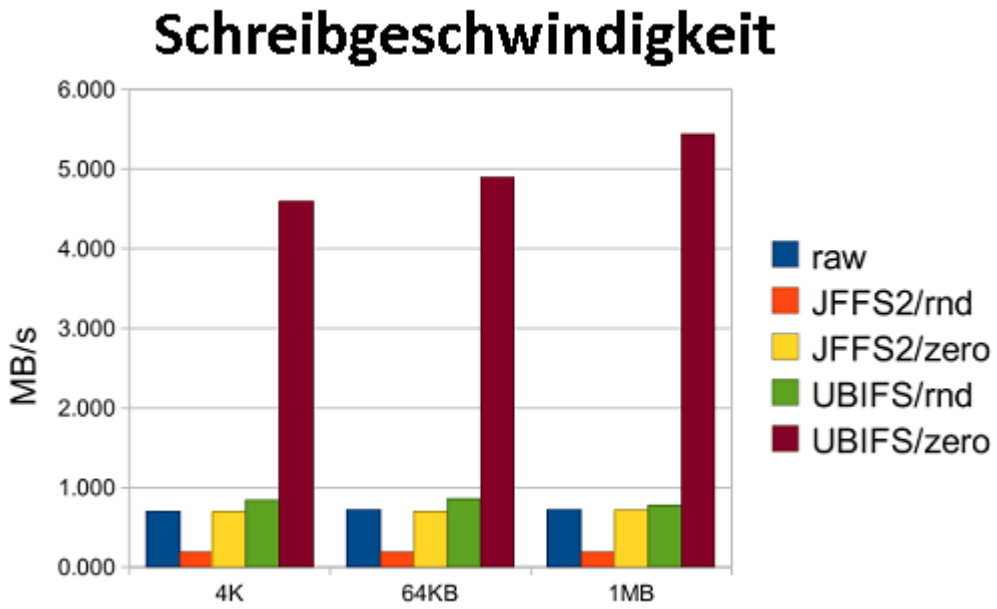


Abbildung 41: Schreibgeschwindigkeit JFFS2 und UBIFS (vgl. [83], Folie 16)

Hier wurde jeweils JFFS2 und UBIFS getestet und jeweils 10 MB in eine Datei in 15 verschiedenen Tests geschrieben. Zunächst wurden diese mit einer (Datei-)Blockgröße von 4 KB geschrieben (Abbildung 41 links), danach mit 64 KB und 1 MB. Die Werte für „raw“ stehen für die Geschwindigkeit, mit der ohne Dateisystem auf den Flashspeicher geschrieben wurde. „rnd“ steht für zufällig generiert Daten, „zero“ für Daten, die nur aus Nullen bestehen. JFFS2 wird bei zufälligen Daten gegenüber der „raw“-Geschwindigkeit langsam, was durch die versuchte Kompression verursacht wird (vgl. [83], Folie 17). Wenn nur Nullen geschrieben werden, ist die Kompression deutlich schneller und erreicht die „raw“-Geschwindigkeit. UBIFS erreicht durch den writeback-Cache auch bei zufälligen Daten eine leicht höhere Geschwindigkeit als „raw“ und JFFS2. Die immensen Werte beim Schreiben der Nullen werden zusätzlich durch die Kompression erreicht. Auch in diesem Test war UBIFS um den Faktor 3 schneller als JFFS2.

In einem weiteren Test, der 2009 vorgestellt wurde (vgl. [84]), zeigte sich, dass YAFFS2 und UBIFS eine etwa gleich schnelle Schreibperformance zeigen. Beide sind schneller als JFFS2, jedoch dort nicht um den Faktor 3, sondern nur um den Faktor 1.5 (vgl. [84], Folie 41). Dies dürfte daran liegen, dass in dem Test die zu schreibenden Daten von /dev/random erzeugt wurden, wodurch die dadurch zusätzlich benötigte CPU-Zeit mit in den Test eingeflossen ist. Die Leseperformance von JFFS2, YAFFS2 und UBIFS war ungefähr auf dem gleichen Niveau. Im Ergebnis zeigte sich, dass UBIFS in nahezu allen Fällen meist besser (und zumindest gleich gut) im Vergleich zu den anderen Flash-Dateisystemen abgeschnitten hat (vgl. [84], Folie 45).

Die Ergebnisse für diese Präsentation (2009) sind damals noch manuell erstellt worden, mit dem Hinweis, dass sie an einer automatisierten Testsuite arbeiten. Mittlerweile steht sie zur Verfügung (vgl. [85]). In den zuletzt unter Linux Kernel 3.1.6 durchgeführten Tests hat sich das Ergebnis bei der Schreibgeschwindigkeit nicht verändert (vgl. [86]). UBIFS und YAFFS2 haben allerdings mittlerweile eine ca. doppelt so hohe Lesegeschwindigkeit wie JFFS2.

Bezogen auf den Datendurchsatz ist von den drei verglichenen Flash-Dateisystemen eindeutig UBIFS die beste Wahl.

5.6.4 Wear Leveling

Die Ergebnisse beim Wear Leveling sind für die verglichenen Dateisysteme sehr unterschiedlich. Ein Vergleich dazu wurde von Homma auf der CELF Linux Konferenz vorgestellt (vgl. [24], Folie 20 ff.). Auf dem Testsystem wurden Partitionen eingerichtet und auf der ersten davon das jeweils zu testende Flashdateisystem eingesetzt. Am Anfang der ersten Partition befanden sich rein statische Daten, die nie ungültig wurden. Die restlichen Daten wurden häufig geschrieben und wieder gelöscht (vgl. Abbildung 42).

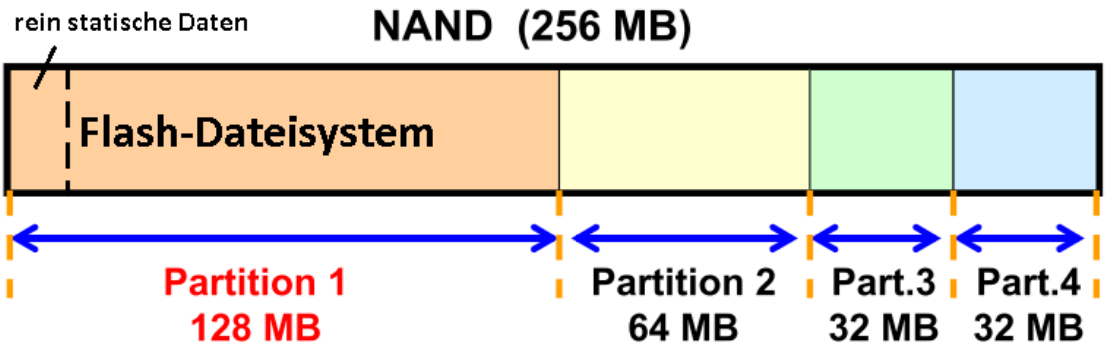


Abbildung 42: Flashspeicheraufteilung für Wear Leveling Vergleich (vgl. [24], Folie 20)

JFFS2 realisiert Wear Leveling, indem der vom Garbage Collector zu säubernde Block nach einer Wahrscheinlichkeit ausgewählt wird. Zu 99% wird ein Block mit mindestens einem ungültigen Datum gewählt, zu 1% einer, der nur gültige Daten enthält. Dadurch werden alle Blöcke einbezogen, jedoch zeigen sich Spitzen im Vergleich der Löschzähler (vgl. Abbildung 43, oben rechts).

YAFFS2 selbst hat kein Wear Leveling implementiert, daher werden Blöcke nur gelöscht, wenn der Garbage Collector sie wählt. Dieser arbeitet allerdings im Greedy-Modus, daher werden Blöcke mit durchweg gültigen Daten niemals gelöscht und nicht mit einbezogen (vgl. Abbildung 43, oben links). Auch die einbezogenen Blöcke zeigen einzelne Ausreißer nach oben und nach unten.

In UBIFS wird das gesamte Wear Leveling von der UBI-Schicht durchgeführt. Dort werden Löschzähler für jeden Block geführt. Sobald die Löschzahl eines Blocks mehr als 5000 von der des am wenigsten gelöschten Block abweicht, werden die Daten der jungen Blocks in den alten Block kopiert, da diese vermutlich statisch sind, sonst wäre der junge Block bereits öfter gelöscht worden. Dieser Schwellwert ist auch konfigurierbar und wurde im Test offenbar auf ca. 20 eingestellt. Es zeigt sich eine sehr gleichmäßige Abnutzung, die auch über Partitionen hinweg durchgeführt wird, da die UBI-Schicht Zugriff auf alle Blöcke hat, im Gegensatz zum darüber liegenden Dateisystem (vgl. Abbildung 43, unten).

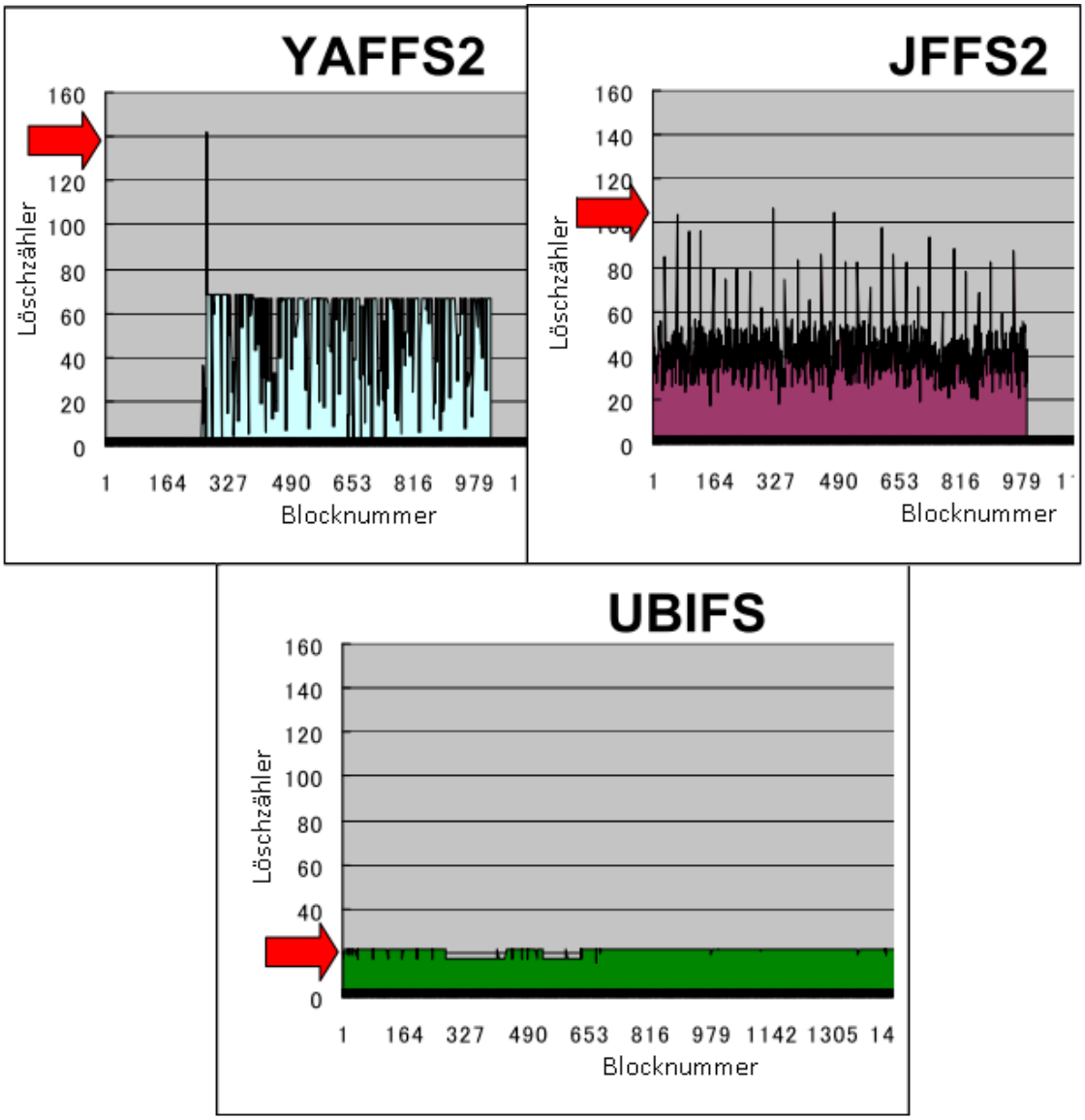


Abbildung 43: Wear Leveling Vergleich: Ergebnisse (vgl. [24], Folie 28)

Auch im Hinblick auf die zu erwartende Lebensdauer des Flashspeichers zeigt sich, dass UBIFS unter den aufgeführten Dateisystemen die beste Option ist.

5.6.5 Übersicht

In Tabelle 3 sind die wichtigsten Leistungs-Eckdaten der neuesten Versionen der vorgestellten Dateisysteme kurz zusammengefasst. Die Beurteilungen ergeben sich aus aus den Ergebnissen der letzten Unterkapitel.

	JFFS2	YAFFS2	UBI+UBIFS	F2FS
Flashspeicher-Art	NOR, NAND	NAND	NOR, NAND	-
Schreiben	mittel	mittel	gut	gut
Lesen	mittel	mittel	gut	gut
Mounttime (sauber)	gut	gut	gut	gut
Mounttime (unsauber)	schlecht	schlecht	gut	gut
Hauptspeicherbedarf	mittel	gut	mittel	?
Wear Leveling	mittel	schlecht	gut	-

Tabelle 3: Übersicht Flash-Dateisysteme

Prinzipiell gibt es keine guten Gründe, noch JFFS oder YAFFS einzusetzen, da UBIFS in jeder Hinsicht überlegen ist, insbesondere bei sehr großen Flashspeichern wegen der Mounttime. Sinnvoll kann YAFFS sein, wenn es auf einem nur von ihm unterstützten Betriebssystem genutzt werden soll. Dem gegenüber steht, dass bei vielen eingebetteten Systemen meist nur eine sehr kleine Partition für das Betriebssystem benötigt ist, z.B. für kabellose Router oder Digitalkameras. Die Erweiterung des Speicherplatzes auf mehrere Gigabyte wird über zusätzliche Flashkarten realisiert, z.B. via eMMC- oder SD-Karten, die üblicherweise nur über eine FTL angesprochen werden können. In dem Einsatzgebiet spricht nichts dagegen, bis zu einem späteren Umstieg weiterhin JFFS oder YAFFS für das Betriebssystem der Geräte zu nutzen.

6 Fazit

Flashspeicher werden sich wegen ihrer unübersehbaren Vorteile gegenüber herkömmlicher Speichertechnik auch zukünftig schnell verbreiten. Die einzig große Hürde sind aktuell noch die Kosten, jedoch fallen die Preise pro Gigabyte, etwa für SSDs, schnell und könnten sich bei kleineren Kapazitäten bereits 2014 an das Niveau von herkömmlichen Festplatten angleichen (vgl. [87]).

Im Bereich der Flash Translation Layer (FTL) werden von Wissenschaftlern immer neue Adressübersetzungs- und Verwaltungsschemata vorgeschlagen und veröffentlicht. Die tatsächlich von den Herstellern genutzten Firmwares gelten jedoch als Geschäftsgeheimnis, es ist daher nicht bekannt, ob und wo die entwickelten FTLs eingesetzt sind. Darauf setzt das F2FS-Dateisystem von Samsung auf. Es ist noch sehr jung, wird aktiv weiterentwickelt und verbessert die Geschwindigkeit auch solcher Geräte, die nur über einen Controller angesprochen werden können.

Bei den Flashmechanismen wie der Garbage Collection oder das Wear Leveling wurden immer weitere Verbesserungen vorgeschlagen, die insbesondere gesteigerte Effizienz bei der Identifizierung und Aufteilung von Daten in *hot* und *cold* erreichen. Die Entwickler der offenen Flash-Dateisysteme zeigen sich davon allerdings unbeeindruckt und setzen auf ihre eigenen Lösungen, die meist auf den sehr alten und einfachen Vorgehensweisen basieren. Dort besteht sicherlich noch Verbesserungspotential.

Die Flash-Dateisysteme selbst haben sich über die letzten 15 Jahre rasant entwickelt und wurden mit jeder neuen Veröffentlichung besser. Wo sich anfangs noch starke Skalierungsprobleme bei etwa JFFS und auch YAFFS gezeigt haben, wurden neue Funktionen eingebaut, die diese Probleme in den meisten Anwendungsszenarien beseitigen. UBIFS wurde Jahre danach mit diesen Erfahrungen im Hinterkopf konzipiert und hat auch diese Schwierigkeiten ordentlich gelöst.

Da die Probleme der frühen Entwicklungsstadien der Flash-Dateisysteme gelöst sind, sollten die eingesetzten Techniken jetzt weiter verbessert werden. Ideen dazu sind in vielen der erklärten Algorithmen und zitierten Veröffentlichungen beschrieben.

7 Literaturverzeichnis

[1] M. Chiang, P. Lee, and R. Chang, “Using data clustering to improve cleaning performance for flash memory,” *Software-Practice and Experience*, vol. 29, no. July 1998, pp. 267–290, 1999.

[2] a. Kolodny, S. T. K. Nieh, B. Eitan, and J. Shappir, “Analysis and modeling of floating-gate EEPROM cells,” *IEEE Transactions on Electron Devices*, vol. 33, no. 6, pp. 835–844, Jun. 1986.

[3] C. Windeck, “SSD-Markt soll dank Ultrabooks kräftig wachsen,” *heise online*, 2013. [Online]. Available: http://www.heise.de/newsticker/meldung/SSD-Markt-soll-dank-Ultrabooks-kraeftig-wachsen-1791247.html. [Accessed: 08-Jul-2013].

[4] G. Gasior, “Crucial’s M500 SSD reviewed,” 2013. [Online]. Available: http://techreport.com/review/24666/crucial-m500-ssd-reviewed/9. [Accessed: 08-Jul-2013].

[5] heise online, “Preisvergleich herkömmlicher Festplatten mit einer Kapazität ab 3 Terabyte,” 2013. [Online]. Available: http://www.heise.de/preisvergleich/?cat=hde7s&xf=958_3000&sort=p. [Accessed: 08-Jul-2013].

[6] heise online, “Preisvergleich von Solid State Drivers mit einer Kapazität ab 240 GB,” 2013. [Online]. Available: http://www.heise.de/preisvergleich/?cat=hdssd&xf=252_245760&sort=p. [Accessed: 08-Jul-2013].

[7] D. Frohman-Bentchkowsky, “Memory behavior in a floating gate avalanche injection MOS (FAMOS) structure,” *Applied Physics Letters*, vol. 18, no. 8, pp. 332–334, 1971.

[8] M. Fujio and I. Hisakazu, “Semiconductor memory device and method for manufacturing the same,” Patent No. 4531203, 1985.

[9] R. Bez, E. Camerlenghi, a. Modelli, and a. Visconti, “Introduction to flash memory,” *Proceedings of the IEEE*, vol. 91, no. 4, pp. 489–502, Apr. 2003.

[10] J. Brewer and M. Gill, *Nonvolatile Memory Technologies with Epmhasis on Flash: A comprehensive Guide to Understanding and Using NVM Devices*. Hoboken: Wiley-IEEE Press, 2011, p. 792.

[11] W. Schiffman, H. Bähring, and U. Hönig, *Technische Informatik 3: Grundlagen der PC-Technologie*. Berlin: Springer, 2011, p. 503.

[12] P. Pavan, R. Bez, P. Olivo, and E. Zanoni, “Flash memory cells-an overview,” *Proceedings of the IEEE*, vol. 85, no. 8, 1997.

[13] L.-P. Chang and L.-C. Huang, “A low-cost wear-leveling algorithm for block-mapping solid-state disks,” *ACM SIGPLAN Notices*, vol. 47, no. 5, p. 31, May 2012.

[14] C. Windeck, “Auch Samsung produziert Flash-Chips mit 16 GByte,” *heise online*, 2013. [Online]. Available: http://www.heise.de/ct/meldung/Auch-Samsung-produziert-Flash-Chips-mit-16-GByte-1840103.html. [Accessed: 08-Jul-2013].

[15] T. Iyer, “Toshiba Announces Availability of 19 nm NAND SSDs,” *Tom’s Hardware*, 2013. [Online]. Available: http://www.tomshardware.com/news/MLC-NAND-mSATA-19nm,21500.html. [Accessed: 08-Jul-2013].

[16] Tech PowerUp, “Samsung Mass Producing 10 nm Class High-Performance 128-Gbit 3-bit MLC NAND Flash,” 2013. [Online]. Available: http://www.techpowerup.com/182650/samsung-mass-producing-10-nm-class-high-performance-128-gbit-3-bit-mlc-nand-flash.html. [Accessed: 08-Jul-2013].

[17] C. Park, J. Seo, S. Bae, H. Kim, S. Kim, and B. Kim, “A low-cost memory architecture with NAND XIP for mobile embedded systems,” in *Proceedings of the 1st IEEE/ACM/IFIP*

international conference on Hardware/software codesign and system synthesis, 2003, pp. 138–143.

[18] C. Lee, S. Baek, and K. Park, “A hybrid flash file system based on nor and nand flash memories for embedded devices,” *Computers, IEEE Transactions on*, vol. 57, no. 7, pp. 1002–1008, 2008.

[19] Micron Technology inc., “NAND Flash 101 : An Introduction to NAND Flash,” 2006. [Online]. Available: http://download.micron.com/pdf/technotes/nand/tn2919.pdf. [Accessed: 14-Apr-2013].

[20] L. Han, Y. Ryu, and K. Yim, “CATA: A Garbage Collection Scheme for Flash,” in *Ubiquitous Intelligence and Computing*, 2006, pp. 103–112.

[21] A. Tal, “Two Technologies Compared: NOR vs. NAND White Paper,” pp. 1–14, 2003.

[22] L.-P. Chang, T.-W. Kuo, and S.-W. Lo, “Real-time garbage collection for flash-memory storage systems of real-time embedded systems,” *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 4, pp. 837–863, Nov. 2004.

[23] L.-P. Chang, “On efficient wear leveling for large-scale flash-memory storage systems,” in *Proceedings of the 2007 ACM symposium on Applied computing*, 2007, pp. 1126–1130.

[24] T. Homma, “Evaluation of Flash File Systems for Large NAND Flash Memory,” in *CELF Embedded Linux Conference*, 2009, p. 33.

[25] E. Spanjer, “Flash Management – Why and How?,” 2009.

[26] J. Cooke, “The Inconvenient Truths of NAND Flash Memory,” 2007. [Online]. Available: https://www.micron.com/~/media/Documents/Products/Presentation/flash_mem_summit_jcooke_inconvenient_truths_nand.pdf.

[27] U. Schneider and D. Werner, *Taschenbuch der Informatik*. München, 2007, p. 837.

[28] U. Kelter, C. Icking, and L. Ma, *Betriebssysteme. Kurs 1802 der FernUniversität in Hagen*. 2012.

[29] V. Prabhakaran, “Analysis and evolution of journaling file systems,” in *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005, pp. 105–120.

[30] J. Corbet, “Barriers and journaling filesystems,” *LWN.NET*, 2008. [Online]. Available: http://lwn.net/Articles/283161/. [Accessed: 08-Jul-2013].

[31] J. Ousterhout and F. Douglis, “Beating the I/O bottleneck: a case for log-structured file systems,” *ACM SIGOPS Operating Systems Review*, vol. 23, no. 1, pp. 11–28, Jan. 1989.

[32] M. Rosenblum and J. Ousterhout, “The design and implementation of a log-structured file system,” *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, 1992.

[33] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan, “File system logging versus clustering: A performance comparison,” *Proceedings of the USENIX 1995 Technical Conference*, 1995.

[34] T. Blackwell, J. Harris, and M. Seltzer, “Heuristic cleaning algorithms in log-structured file systems,” *Proceedings of the 1995 Winter USENIX*, 1995.

[35] Micron Technology inc., “Garbage Collection in Single-Level Cell NAND Flash Memory,” Boise, 2011.

[36] T. Kgil, D. Roberts, and T. Mudge, “Improving NAND Flash Based Disk Caches,” *2008 International Symposium on Computer Architecture*, pp. 327–338, Jun. 2008.

[37] I. Iliadis, “Performance of the Greedy Garbage-Collection Scheme in Flash-Based Solid-State Drives,” Zürich, 2010.

[38] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. Tauber, “Storage alternatives for mobile computers,” *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pp. 1–14, 1994.

[39] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, “Efficient identification of hot data for flash memory storage systems,” *ACM Transactions on Storage*, vol. 2, no. 1, pp. 22–40, Feb. 2006.

[40] D. Park and D. H. C. Du, “Hot data identification for flash-based storage systems using multiple bloom filters,” *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–11, May 2011.

[41] M. Chiang, P. Lee, and R. Chang, “Managing flash memory in personal communication devices,” in *Proceedings of 1997 IEEE International Symposium on. IEEE*, 1997, pp. 177–182.

[42] L. Chang and T. Kuo, “An adaptive striping architecture for flash memory storage systems of embedded systems,” *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 187–196, 2002.

[43] M. Wu and W. Zwaenepoel, “eNVy: a non-volatile, main memory storage system,” *ACM SigPlan Notices*, pp. 86–97, 1994.

[44] A. Kawaguchi, S. Nishioka, and H. Motoda, “A flash-memory based file system,” in *Proceedings of the Winter 1995 USENIX Technical Conference*, 1995, vol. 1, pp. 1–10.

[45] L. Han, Y. Ryu, T. Chung, M. Lee, and S. Hong, “An intelligent garbage collection algorithm for flash memory storages,” in *Proceedings of the 6th international conference on Computational Science and Its Applications*, 2006, vol. 1, pp. 1019–1027.

[46] S. Oral, G. M. Shipman, and D. a. Dillow, “Harmonia: A globally coordinated garbage collector for arrays of Solid-State Drives,” *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, May 2011.

[47] Y. Kim, J. Lee, S. Oral, D. Dillow, F. Wang, and G. Shipman, “Coordinating Garbage Collection for Arrays of Solid-state Drives,” *IEEE Transactions on Computers*, no. 99, pp. 1–14, 2012.

[48] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, “Design tradeoffs for SSD performance,” in *Proceedings of the USENIX 2008 Technical Conference*, 2008, vol. 08, no. June, pp. 57–70.

[49] S. Di Carlo, M. Fabiano, P. Prinetto, and M. Caramia, “Design Issues and Challenges of File Systems for Flash Memories,” p. 30, 2011.

[50] H. Kim and S. Lee, “An effective flash memory manager for reliable flash memory space management,” *IEICE Transactions on Information and Systems*, vol. E85-D, no. 6, pp. 950–964, 2002.

[51] L. Chang and T. Kuo, “An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems Categories and Subject Descriptors,” 2004.

[52] I. Spectrum, “Flash Memory Survives 100 Million Cycles,” 2012. [Online]. Available: http://m.spectrum.ieee.org/semiconductors/memory/flash-memory-survives-100-million-cycles. [Accessed: 08-Jul-2013].

[53] Z. Wang, M. Karpovsky, and A. Joshi, “Nonlinear multi-error correction codes for reliable MLC NAND flash memories,” *IEEE Transaction on Very Large Scale Integration Systems*, vol. 20, no. 7, pp. 1221–1234, 2012.

[54] A. Hunter and A. Bityutskiy, “UBIFS file system Presentation,” 2008. [Online]. Available: http://www.linux-mtd.infradead.org/doc/ubifs.pdf. [Accessed: 17-Jul-2013].

[55] L. Grupp, “Research Statement,” 2012.

[56] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, “LAST: locality-aware sector translation for NAND flash memory-based storage systems,” *ACM SIGOPS Operating Systems Review*, pp. 36–42, 2008.

[57] M. Chiao and D. Chang, “ROSE: A Novel Flash Translation Layer for NAND Flash Memory Based on Hybrid Address Translation,” *IEEE Transactions on Computers*, vol. 60, no. 6, pp. 753–766, Jun. 2011.

[58] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, “A space-efficient flash translation layer for CompactFlash systems,” *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, pp. 366–375, May 2002.

[59] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, “A superblock-based flash translation layer for NAND flash memory,” *Proceedings of the 6th ACM & IEEE International conference on Embedded software - EMSOFT ’06*, p. 161, 2006.

[60] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, “A log buffer-based flash translation layer using fully-associative sector translation,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 3, pp. 18–55, Jul. 2007.

[61] P. Barrett, S. Quinn, and R. Lipe, “System for updating data stored on a ash EPROM based upon predetermined bit value of indicating pointers,” Patent No. 5392427, 1993.

[62] D. Woodhouse, “JFFS: The journalling flash file system,” in *Ottawa Linux Symposium*, 2001.

[63] Memory Technology Devices Group, “General MTD Documentation,” 2008. [Online]. Available: http://www.linux-mtd.infradead.org/doc/general.html. [Accessed: 16-Jul-2013].

[64] OpenWrt, “OpenWRT Website,” 2013. [Online]. Available: https://openwrt.org/. [Accessed: 20-Jul-2013].

[65] A. Korolev, “Improving JFFS2 RAM usage and performance,” in *The CELF Embedded Linux Conference - Europe*, 2007.

[66] L. Boschetti, “Software Profile: Journaling Flash File System, Version 2 (JFFS2),” 2011.

[67] University of Szeged, “JFFS2 improvement project Website,” 2006. [Online]. Available: http://www.inf.u-szeged.hu/jffs2/mount.php. [Accessed: 21-Jul-2013].

[68] S. Liu, X. Guan, D. Tong, and X. Cheng, “Analysis and comparison of NAND flash specific file systems,” *Chinese Journal of Electronics*, vol. 19, no. 3, pp. 403–408, 2010.

[69] Memory Technology Devices Group, “JFFS2 - FAQ,” 2009. [Online]. Available: http://www.linux-mtd.infradead.org/faq/jffs2.html. [Accessed: 11-Jul-2013].

[70] C. Manning, “How YAFFS works.” p. 25, 2012.

[71] Wookey, “Yaffs - A NAND Flash File System,” in *The CELF Embedded Linux Conference - Europe*, 2007.

[72] Memory Technology Devices Group, “UBI File-System Website,” 2008. [Online]. Available: http://www.linux-mtd.infradead.org/doc/ubifs.html. [Accessed: 21-Jul-2013].

[73] Memory Technology Devices Group, “UBI - Unsorted Block Images Website,” 2009. [Online]. Available: http://www.linux-mtd.infradead.org/doc/ubi.html. [Accessed: 08-Jul-2013].

[74] T. Gleixner, F. Haverkamp, and A. Bityutskiy, “UBI - Unsorted Block Images,” 2006. [Online]. Available: http://linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf. [Accessed: 08-Jul-2013].

[75] A. Hunter, “A Brief Introduction to the Design of UBIFS,” 2008.

[76] C. Egger, “File systems for flash devices,” 2010.

[77] J. Jung, “Application of UBIFS for Embedded Linux Products,” in *LinuxCon Japan*, 2010.

[78] J. Kim, “Flash-Friendly File System (F2FS),” in *Korea Linux Forum*, 2012.

[79] N. Brown, “An f2fs teardown,” *LWN.NET*, 2012. [Online]. Available: http://lwn.net/Articles/518988/. [Accessed: 08-Jul-2013].

[80] L. Romanovsky, “Flash Friendly File System (F2FS) Overview.” 2012.

[81] J.-Y. Hwang, “Flash-Friendly File System (F2FS),” 2013.

[82] W. Norcott, “IOzone Filesystem Benchmark Website.” [Online]. Available: http://www.iozone.org. [Accessed: 30-Jul-2013].

[83] C. Simmonds, “Linux flash file systems - JFFS2 vs. UBIFS,” in *Embedded Systems Conference UK*, 2009.

[84] M. Opdenacker, “Update on filesystems for flash storage,” *free-electrons.com*, pp. 1–53, 2009.

[85] elinux.org, “Flash Filesystem Benchmarks Protocol Website,” 2011. [Online]. Available: http://elinux.org/Flash_Filesystem_Benchmarks_Protocol. [Accessed: 27-Jul-2013].

[86] elinux.org, “Flash Filesystem Benchmarks 3.1 Website,” 2012. [Online]. Available: http://elinux.org/Flash_Filesystem_Benchmarks_3.1. [Accessed: 26-Jul-2013].

[87] Storage Newsletter, “When Will SSD Have Same Price as HDD,” 2013. [Online]. Available: http://www.storagenewsletter.com/news/marketreport/when-will-ssd-have-same-price-as-hdd-priceg2. [Accessed: 07-Aug-2013].

[88] Tae-Sun Chung, T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, “A survey of Flash Translation Layer,” *Journal of Systems Architecture*, vol. 55, no. 5–6, pp. 332–343, 2009.

[89] S. Lim and K. Park, “An efficient NAND flash file system for flash memory storage,” *Computers, IEEE Transactions on*, vol. 55, no. 7, pp. 906–912, 2006.

[90] W. K. Josephson, “A Direct-Access File System for a New Generation of Flash Memory,” Princeton University, 2011.