Douglas Cunningham, Petra Hofstedt, Klaus Meer, Ingo Schmitt (Hrsg.): INFORMATIK 2015
Lecture Notes in Informatics (LNI), Gesellschaft f¨ur Informatik, Bonn 2015

Softwarebasierte Fehlertolerenz für Flash-Speicher von mikrocontroller-basierten Systemen

Patryk Skoncej[1] Felix Mühlbauer[2] Mario Schölzel[3]

Abstract: In diesem Papier wird eine softwarebasierte Methode für die Handhabung von permanenten Fehlern in kostengünstigen Speichern von eingebetteten Systemen vorgestellt. Im Wesentlichen wird der Programmcode derart angepasst, dass keine fehlerhaften Speicherzellen mehr benutzt werden. So ist es möglich permanente Fehler in einfachen und günstigen Speicherbausteinen ohne zusätzlichen Hardwareaufwand zu tolerieren. Dies ist insbesondere für langlebige kostengünstige eingebettete Systeme, z. B. Sensorknoten, interessant, die solche Speicher einsetzen. Verschiedene Methoden für die Handhabung von Speicherfehlern im Daten- und Programmbereich werden betrachtet. Die Methoden für den Programmbereich werden für Flash-Speicher evaluiert. Basierend auf echten Testdaten aus der Produktion von Flash-Speichern wird gezeigt, dass mit der softwarebasierten Methode ein signifikanter Hardwareanteil eingespart werden kann, ohne die Fehlertoleranz von Speicherfehler zu verlieren.

Keywords: Fehlertolerante Systeme, Flash-Speicher, Softwarebasierte Selbstreparatur, Yield-Improvement

1 Einleitung

Die immer weiter schrumpfende Strukturgröße von CMOS-Bausteinen führt zu einer steigenden Verbreitung von eingebetteten elektronischen Geräten, weil mehr Funktionalität bei geringeren Kosten angeboten werden kann. Dies erlaubt die Herstellung von komplexen eingebetteten Systemen mit leistungsstarken CPUs, wie sie zum Beispiel in Steueranlagen von Antriebssystemen im Automobil- und Luftfahrtsektor eingesetzt werden. In solchen Systemen werden etablierte und kostspielige Fehlertoleranzmethoden, wie DMR oder TMR (dual/triple modular redundancy), eingesetzt. Andererseits steigt der Bedarf an kostengünstigen eingebetteten Systemen für das Internet der Dinge und die Industrie 4.0 Infrastruktur. Mit Funkkommunikation ausgestattete Mikrocontroller befinden sich in vielen SmartHome-Anwendungen, wie z. B. Leuchten, Thermostaten und anderen Geräten, die ein Netzwerk bilden, um ihre Umgebung zu Erfassen und zu Steuern. Solche Technologien werden auch eingesetzt, um Produktionsanlagen zu Überwachen und zu Steuern oder für die lebenslange Überwachung von Geräten. Dazu müssen zwei gegensätzliche Bedingungen erfüllt werden: Einerseits müssen die eingesetzten eingebetteten Systeme so kostengünstig wie möglich sein. Zum Beispiel werden Leuchtmittel mit Elektronik entsorgt, sobald sie kaputt gehen. Die eingebaute Elektronik muss folglich kostengünstig sein. Andererseits sollte die eingebaute Elektronik nicht die Hauptursache von Fehlern

[1] IHP Frankfurt (Oder) und BTU Cottbus-Senftenberg, skoncej@ihp-microelectronics.com
[2] Universität Potsdam, muehlbauer@cs.uni-potsdam.de
[3] IHP Frankfurt (Oder) und Universität Potsdam, schoelzel@ihp-microelectronics.com

sein. Zum Beispiel sollte ein Leuchtmittel nicht deshalb kaputt gehen, weil der Mikrocontroller für die Funkkommunikation versagt. Leider steigt die Wahrscheinlichkeit von permanenten Fehlern in CMOS-Bausteinen mit der Verkleinerung der Strukturen [CBT09]. Folglich müssen eingebettete Systeme, so bald sie vermehrt für das Internet der Dinge und Industrie 4.0 eingesetzt werden, eine lange Lebenszeit erreichen und einfache und kostengünstige Komponenten haben. Dafür werden entsprechende Redundanztechniken benötigt. Gewöhnlich wird dies nicht durch das Hinzufügen von Hardwareredundanz (DMR, TMR) erreicht. Eine Lösung könnten softwarebasierte Verfahren sein, indem ein Teil der inhärent verfügbaren Redundanz in diesen eingebetteten Systemen genutzt wird, ohne Extrahardware zu benötigen. Die kleinen eingebetteten Systeme, die in diesem Papier betrachtet werden, sind typischerweise aus sehr einfachen 8- oder 16-Bit Mikrocontrollern mit etwas Speicher und Peripherie aufgebaut. Die Speicher sind eine Mischung aus ROM, RAM und Flash-Speicher mit manchmal nicht mehr als insgesamt 64 kByte. Dennoch belegen die Speicher einen signifikanten Bereich auf dem Chip verglichen mit der CPU. Abbildung 1 zeigt diese Verteilung für einen Mikrocontroller aus der MSP430-Familie mit 64 kByte Flash und 16 kByte RAM.

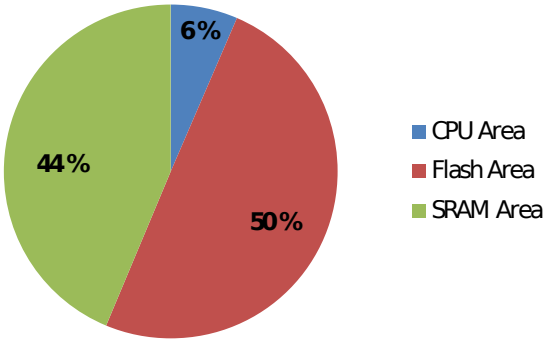

Abb. 1: Belegung der Chipfläche

95 % der Chipfläche wird für Speicher benutzt. Folglich würde eine fehlertolerante CPU die Zuverlässigkeit des Systems nicht signifikant erhöhen, unter der Annahme einer gleich verteilten Fehlerrate. Statt dessen müssen Fehler in den Speichern behandelt werden. Typischerweise wird nicht der komplette Speicher von einer Anwendung verwendet und es verbleiben ungenutzte Speicherwörter. Diese Redundanz wird von den bekannten hardwarebasierten Methoden für permanente Fehler nicht ausgenutzt. Dieses Papier stellt eine softwarebasierte Methode für Prozessoren vor, die sich diese inhärente Redundanz zu Nutze macht. Die Anwendung wird derart im Speicher neu verteilt, dass keine fehlerhaften Speicherzellen mehr benutzt werden. Dies ist möglich, weil die Speicherhierarchie von solchen Systemen sehr einfach ist und z. B. keine Caches oder virtuelle Adressierung zum Einsatz kommt. Die Effizienz dieser Strategie wird für verschiedenen Benchmark-Programme unter Verwendung echter Fehlerdaten aus der Produktion von Flash-Speichern evaluiert.

2 Verwandte Arbeiten

Wegen der sehr regelmäßigen Struktur von Halbleiterspeichern wird Fehlertoleranz typischerweise durch hardwarebasierte Methoden realisiert. Die beiden bekanntesten Ansätze sind: 1) Die Maskierung von Fehlern durch Fehlerkorrigierende Blockcodes (ECC) und 2) Aktive Redundanztechniken die defekte Zellen austauschen.

Das Hauptziel von ECC-basierten Methoden ist die Korrektur von transienten und/oder strahlungsverursachten Speicherfehlern. Die bekanntesten Klassen von Codes sind erweiterte Hamming/Hsiao Codes für Einzelbitfehler und Reed-Solomon (RS) und Bose-Chaudhuri-Hocquengham (BCH) Codes für Mehrbitfehler [SS11]. Mehrbit-ECC Codes erhöhen oft Chipfläche, Laufzeit und Energiekosten, was in kleinen eingebetteten Anwendungen schwer zu akzeptieren ist [AIB10, Ch12]. Deshalb werden für eingebettete Speicher, mit Beschränkungen bezüglich erlaubter Latenzzeit und Chipfläche, bevorzugt SECDED Codes (single error correction and double error detection), wie der erweiterte Hamming/Hsiao Code, eingesetzt. Während ECC-basierte Methoden hauptsächlich für die Maskierung von temporären Fehlern Verwendung finden, zielen aktive Redundanztechniken auf permanente Fehler ab und dienen hauptsächlich der Erhöhung der Fertigungsausbeute. Die geläufigste Implementierung von aktiver Redundanz in eingebetteten Speichern ist das BISR (build-in self-repair) Schema, welches sehr ähnlich für RAM [Li03] und Flash-Speicher [HCW06, HCW10] ist. Die Redundanz kann grundsätzlich in drei Klassen eingeteilt werden: Reserveblöcke, Reservezeilen und/oder Reservespalten. Die Reserven werden von einem BISR-Design verwaltet, welches aus drei Funktionen besteht: 1) Eine BISR-Funktion für die Erkennung von fehlerhaften Speicherelementen 2) Die BIRA (build-in redundancy analysis) bestimmt eine passende Reserve 3) Das AR (address reconfiguration) Modul wird benötigt, um Speicherzugriffe entsprechend zu den Reserveelementen umzuleiten [HCW10, HCW07]. Die Komplexität des BISR-Designs hängt hauptsächlich von der BIRA-Funktion ab, die versucht Reserveelemente effizient zu vergeben. Sowohl die Hardwarekomplexität als auch die Laufzeit der BIRA-Funktion beeinflussen die für eine Reparatur benötigte Art und Anzahl von Reserveelementen. Dabei kann ein blockbasiertes Ersetzen von defekten Speicherbereichen effektiver sein als eine zeilen- oder spaltenbasierte Reparatur. Der Nachteil kann allerdings ein höherer Flächenbedarf auf dem Chip und größere Zuordnungstabellen sein [HCW10, Pe08]. Allein mit der Zeilen- oder der Spaltenredundanz (1D) kann kaum eine hohe Reparaturgeschwindigkeit erreicht werden. Dagegen können durch die Kombination der beiden (2D) sehr hohe Geschwindigkeiten erreicht werden, jedoch ist die 2D-Analyse NP-vollständig, was komplexe BIRA-Algorithmen zur Folge hat [HCW06, HCW07]. Folglich sind die kosteneffizientesten Lösungen für die Reparatur von permanenten Fehlern in kleinen eingebetteten Speicher entweder blockbasierte oder eindimensionale Reparaturverfahren.

Allerdings wurden bisher kaum softwarebasierte Reparaturverfahren für Speicher untersucht. Eine Möglichkeit ist es Fehlertoleranz-Methoden für Hardware in Software zu Implementieren, die typischerweise auf temporäre Fehler abzielen [Li06, Re99]. Solche Ansätze basieren auf der redundanten Ausführung von Befehlen und die Verdoppelung von Daten auf Softwareebene. Durch die Vervielfältigung von Daten können auch permanente Fehler in Datenspeichern erkannt werden. Allerdings wird so auch die benötigte Speicher-

größe verdoppelt und Fehler im Programmspeicher werden nicht berücksichtigt. Um diese Methode auf den Programmspeicher anzuwenden, muss das Programm verdoppelt werden oder der Kontrollfluss muss per Software überprüft werden [Go03, OSM00]. Cheng und Gupta beschreiben eine softwarebasierte Idee für die Handhabung von permanenten Fehlern in der fehlerhafte Speicherzellen vermieden werden [CG11]. Das vorgestellte Verfahren ist sehr ähnlich zu dem in [Sc10] für Prozessoren. Allerdings wird kein Beweis angeboten, dass echte Speicherfehler repariert werden können. Darüber hinaus wird keine Methode zur Vermeidung von fehlerhaften Speicherzellen diskutiert. Beide Aspekte werden im vorliegenden Papier untersucht.

3 Softwarebasierte Fehlerbehandlung in Speichern

Der Speicher wird im Folgenden aus einer funktionalen Sicht, d. h. als ein Vektor von Datenwörtern, betrachtet. Ein physikalischer Defekt kann eines, mehrere oder alle Datenwörter beeinträchtigen und in einem Wort wiederum, eine beliebige Anzahl von Bits. Das grundlegende Konzept für eine softwarebasierte Selbstreparatur von Speichern basiert auf der Methode für Prozessoren in [Sc10, Sc11]. Die inhärent verfügbare Redundanz in den Datenwörtern wird per Software verwaltet. Das bedeutet, dass die Verwendung fehlerhafter Datenwörter vermieden wird, in dem die Anwendung im Programmspeicher an den aktuellen Fehlerzustand des Speichers angepasst wird. Das Programm wird nach dem Herstellungsprozess angepasst, sobald der Flash-Speicher programmiert werden soll. Für die Anpassung wird eine Fehlertabelle des Speichers benötigt. Diese Tabelle enthält die Adressen aller fehlerhaften Datenwörter, die nicht mehr verwendet werden sollen. Ein Speichertest, der nach der Produktion ausgeführt wurde, könnte solche Daten liefern. Die Anwendung wird basierend auf der Tabelle an den Fehlerzustand des Speichers angepasst. Dabei muss zwischen Fehlern die das Codesegment betreffen von Fehlern die das Datensegment betreffen unterschieden werden. Beide Fälle werden in den folgenden Abschnitten untersucht.

3.1 Fehlerbehandlung in Datenspeichern

Das Datensegment wird im RAM abgelegt. Viele kleine eingebettete Systeme sind nur mit ein paar Kilobytes RAM ausgestattet. Dieser Speicher wird benutzt für

- die Speicherung von globale Daten,
- die Organisation der dynamischen Speicherverwaltung (heap) und
- die Implementierung des Stapelspeichers (stack).

Für jeden dieser drei Fälle wird eine andere Vorgehensweise im Umgang mit defekten Datenwörtern benötigt. Sie werden in den folgenden Abschnitten beschrieben.

3.1.1 Globale Daten

Im Bereich für globale Daten werden globale Variablen und Konstanten für die Laufzeit der Anwendung gespeichert. Die Adressen werden während der Verlinkung der Anwendung zugewiesen und der Zugriff auf diese Elemente erfolgt typischerweise über absolute Adressen. Ist ein Element von einem Speicherfehler betroffen, muss es an eine andere Speicheradresse verschoben werden. Dies erfordert eine Anpassung aller zugehörigen Speicherzugriffe im Programmcode. Basierend auf dem Binärcode der Anwendung ist dies sehr schwierig, weil eine differenzierte Datenflussanalyse erforderlich ist, um alle Zugriffe auf den globalen Speicher zu identifizieren. Einfacher ist es die Daten des Linkers zu benutzen. Dort sind alle globalen Symbole und Referenzen auf diese verfügbar. Alle notwendigen Informationen können aus den Linker-Dateien ausgelesen werden und damit ist eine Verschiebung von globalen Symbolen durchführbar.

3.1.2 Dynamische Speicherverwaltung

Auf dem Heap werden dynamisch angeforderte Daten gespeichert. Die dynamische Speicherverwaltung muss entweder vom Betriebssystem oder einer eigenen malloc-Funktion implementiert werden. In beiden Fällen muss die Fehlertabelle des Speichers zur Laufzeit der Anwendung zur Verfügung stehen. Mit Hilfe der Tabelle können fehlerhafte Speicherbereiche bei einer Speicheranfrage übersprungen werden.

Bei einer selbst-implementierten malloc-Funktion kann die Fehlertabelle direkt benutzt werden und in die Speicherverwaltung einfließen. Wenn ein Betriebssystem zum Einsatz kommt kann die malloc-Funktion möglicherweise nicht verändert werden. In diesem Fall, kann die Anwendung jedoch kleine Speicherblöcke anfordern bis der Speicher voll ist, um anschließend alle fehlerfreien Blöcke, und nur diese, wieder freizugeben.

3.1.3 Stapelspeicher

Der Stapelspeicher wird benutzt um lokale Variablen, Parameter und Rückgabewerte von Funktionsaufrufen zu Speichern. Für eingebettete Systeme kann die benötigte Größe des Stapelspeichers oft sehr genau abgeschätzt werden, weil in den meisten Fällen keine rekursiven Funktionen eingesetzt werden. Durch eine Analyse des Aufrufgraphen der Anwendung kann, basierend auf den Rahmengrößen der Funktionen, die maximale Größe des Stapelspeichers bestimmt werden. Die Rahmengröße einer Funktion ist dabei der Bedarf an Stapelspeicher für ihren Aufruf. Mit diesem Wissen kann ermittelt werden, ob der Stapelspeicher fehlerhafte Speicherwörter enthält. Zum Beispiel ist es sehr gewöhnlich den Stapelspeicher an der höchsten Adresse des RAMs beginnen zu lassen. Der Stapelspeicher wächst dann in Richtung kleinerer Adressen. Wenn dieser Speicherbereich fehlerhaft ist, kann der ganze Stapelspeicher in einen andern zusammenhängenden fehlerfreien Speicherbereich verschoben werden. Die Verschiebung ist sehr einfach, weil in eingebetteten Systemen der Stapelzeiger von der Anwendung selbst initialisiert wird und hier eine ande-

re Startadresse genutzt werden kann. Jedoch benötigt diese Methode einen ausreichend großen, zusammenhängenden und fehlerfreien Speicherbereich. Ferner ist möglicherweise auch eine Verschiebung der globalen Daten oder des Heapspeichers notwendig. Zum Beispiel können globale Variablen vor und nach dem Stapelspeicher angelegt werden, um den verfügbaren Speicher effizient zu nutzen.

Für fragmentierte Stapelspeicher dagegen, ist es möglich Prolog und Epilog der Funktionen anzupassen. Im Prolog wird sofort der für die Funktion benötigte Speicher reserviert. Typischerweise geschieht das durch eine Verringerung des Stapelzeigers um eine Konstante. Ist dieser Speicherbereich fehlerhaft, kann der Stapelzeiger entsprechend weiter verringert werden, bis das nicht mehr der Fall ist. Im Epilog wird der belegte Speicherbereich wieder freigegeben. Diese Modifikation erfordert eine Anpassung des Compilers und die Fehlertabelle muss für die dynamische Überprüfung im Prolog zur Laufzeit zur Verfügung stehen.

3.1.4 Zusammenfassung

Mit der vorgestellten Methode ist es möglich Fehler im RAM von eingebetteten Systemen zu handhaben. Für die dynamische Speicherverwaltung und für die effiziente Nutzung eines Speicherbereichs als Stapelspeicher ist eine Fehlertabelle des Speichers zur Laufzeit notwendig. Die vorgestellten Methoden setzen jedoch eine Anpassung der Toolchain (Compiler und Linker) voraus.

Im nächsten Abschnitt wird gezeigt, das für den Programmbereich keine Anpassung der Toolchain notwendig ist.

3.2 Fehlerbehandlung im Programmspeicher

Der Programmcode ist typischerweise im ROM oder in einem Flash-Speicher. Im folgenden werden nur Flash-Speicher betrachtet, weil der Code in ROMs nicht verändert werden kann.

In der Regel wird für eine Anwendung in kleinen eingebetteten Systemen nur ein Binary erzeugt, welches alle Funktionen nacheinander enthält (siehe Abbildung 2 a, Funktion F1 bis F4).

Um die Benutzung von fehlerhaften Speicherwörtern für Programmcode zu vermeiden, werden zwei Wege vorgeschlagen:

- Fehlerhafte Speicherwörter innerhalb einer Funktion sind erlaubt
- Fehlerhafte Speicherwörter sind nur außerhalb von Funktionen erlaubt

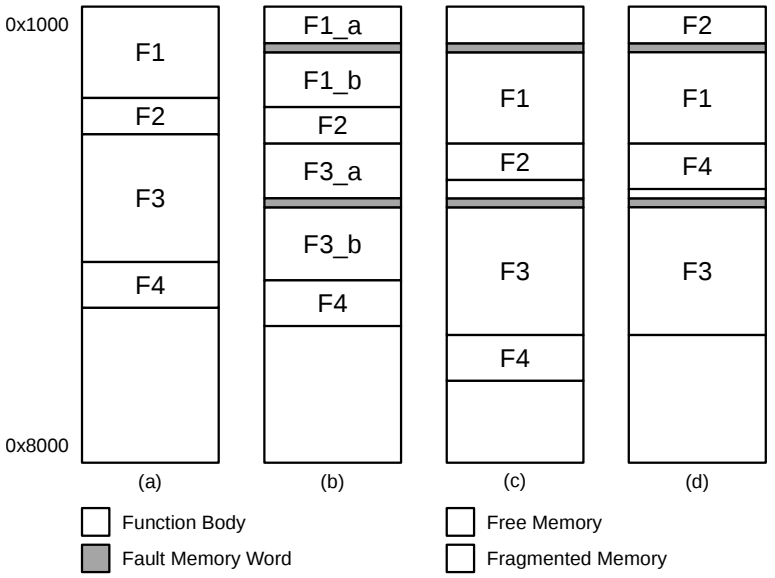

Abb. 2: Fehlerbehandlung im Programmspeicher: (a) Nutzung eines fehlerfreien Speichers (b) Fehlerstellen werden übersprungen (c) Funktionen werden in gleicher Reihenfolge neu platziert (d) Funktionen werden neu verteilt.

3.2.1 Methode 1: Zerteilung von Funktionen

Abbildung 2 b zeigt ein Beispiel für fehlerhafte Speicherwörter innerhalb einer Funktion. Eine Fehlerstelle wird hier durch einen unbedingten Sprungbefehl, der unmittelbar davor eingebaut wird, übersprungen. Auf diese Weise ist eine sehr feingranulare Fehlerbehandlung möglich und es wird nur wenig Speicher verschwendet. Auf der anderen Seite werden Funktionen zerteilt, wie z. B. F1 und F3. Dies erfordert eine Anpassung der verschiedenen Branch-Befehle innerhalb der Funktion. Zum Beispiel verwenden viele Prozessorarchitekturen eine PC-relative Adressierung in bedingten Sprungbefehlen. Sobald ein Sprungbefehl ein fehlerhaftes Speicherwort überspringen würde, muss dieser Befehl angepasst werden. In [MSV13] wurde gezeigt, dass dies sogar auf der Grundlage des Programmcodes möglich, jedoch sehr rechenintensiv, ist.

3.2.2 Methode 2: Neuverteilung von Funktionen

Eine andere Möglichkeit ist es die Funktionen neu anzuordnen ohne deren Implementierung zu verändern, d. h. Leerstellen können sich nur zwischen den Funktionen befinden. Dafür gibt es zwei Möglichkeiten:

- Die Reihenfolge der Funktionen bleibt erhalten (Abbildung 2 c) oder
- die Reihenfolge der Funktionen darf verändert werden (Abbildung 2 d).

Im zweiten Fall kann das Ausmaß an unbenutzten Speicherfragmenten minimiert werden, indem jene Funktionen ausgewählt werden die am besten in eine der Lücken zwischen zwei Fehlerwörtern passen. Dies ist das bekannte Behälterproblem (bin packing problem). Es ist NP-schwer und kann jedoch z. B. mit einfachen heuristischen Methoden gelöst werden.

Ein einfacherer Algorithmus kann benutzt werden, indem die Reihenfolge der Funktionen beibehalten wird. Dies kann jedoch eine starke Fragmentierung nach sich ziehen, wie in Abbildung 2 c dargestellt. Mit dieser Methode müssen nur die Befehle für Funktionsaufrufe angepasst werden, was die Anpassung des Programms vereinfacht.

4 Ergebnisse

In diesem Abschnitt wird das Potential der vorgestellten softwarebasierten Fehlerbehandlung an Hand von echten Produktionsfehlern in Flash-Speichern evaluiert.

4.1 Produktionsausbeute bei Flash-Speichern

Die Evaluation basiert auf echten Testdaten von Flash-Speichern in 250 nm-Technologie. Zwei Versionen werden betrachtet: 8K und 16K. Die 8K-Speicher haben 8192 32-Bit Datenwörter und damit 32 kByte und die 16K-Version 16384 Wörter und damit 64 kByte. Es stehen die Testergebnisse von je 244 Speichern der beiden Varianten, verteilt auf vier Wavern, zur Verfügung. Sie sind in Tabelle 1 zusammengefasst.

Speicherart	Anzahl	Fehlerfrei	Fehlerhaft	Ausbeute
8K	244	210	34	86,1 %
16K	244	195	49	79,9 %

Tab. 1: Produktionsausbeute für 8K- und 16K-Flash-Speicher

Die 8K-Version hat mit 86,1 % eine höhere Ausbeute als die 16K-Variante mit nur 79,9 %. Eine detaillierte Diagnose zeigte, dass in einem Speicher meistens nur wenige Datenwörter defekt sind. In Abbildung 3 ist jeweils die Anzahl der Speicher mit einer bestimmten Anzahl an Fehlerwörtern dargestellt.

Nur bei 14 der 34 defekten Speicher sind alle Datenwörter beschädigt. Ein Speicher ist größtenteils unbrauchbar. In solchen Bausteinen sind typischerweise die Daten- und/oder Adressleitungen fehlerhaft, d. h. nur 6 % der 244 Bausteine habe Defekte die sie unbrauchbar machen. Bei 10 % (19 Stück) sind weniger als 446 Datenwörter beschädigt, d. h. nur weniger als 5,5 % des Speichers kann nicht mehr benutzt werden. Diese Speicher können für Anwendungen eingesetzt werden, die diese kleine Speichereinbusen hinnehmen und die Benutzung der fehlerhaften Speicherwörter vermieden werden kann.

Eine ähnliche Betrachtung wurde für die 16K-Speicher durchgeführt (siehe Abbildung 4). Bei 16 der 49 Bausteinen sind knapp 25 % oder mehr der Speicherwörter betroffen. Andererseits sind in den übrigen 33 beschädigten Speichern nur weniger als 1 % der Wörter

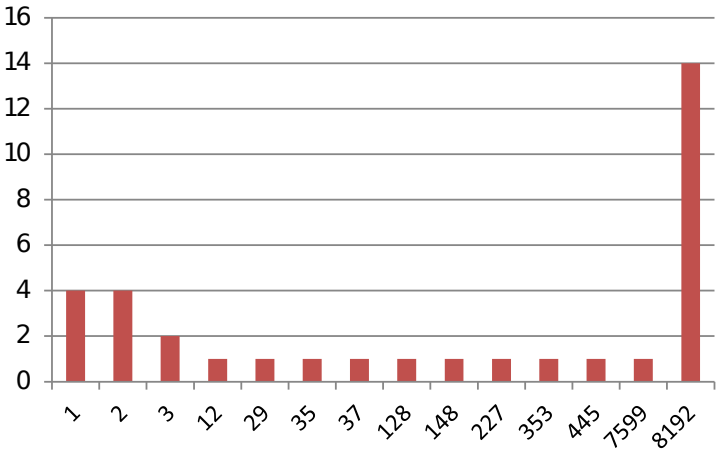

Abb. 3: Verteilung der Fehlerhäufigkeit für 8K-Speicher

fehlerhaft. Auch diese Speicher sind potentielle Kandidaten für Anwendung mit Fehlerbehandlung.

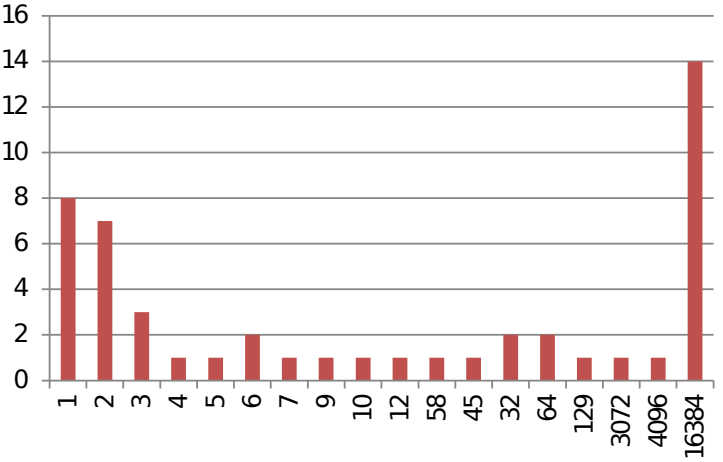

Abb. 4: Verteilung der Fehlerhäufigkeit für 16K-Speicher

4.2 Verbesserung der Produktionsausbeute

Im Folgenden wird dargestellt in wie fern die Produktionsausbeute für die oben beschriebenen Flash-Speicher unter Anwendung der vorgestellten Fehlerbehandlungsmethoden verbessert werden kann. Bei der softwarebasierten Reparatur ist die erreichbare Verbesserung vom Grad der Speichernutzung der Anwendung und der einzelnen Funktionen abhängig. Deshalb werden im Folgenden Diagnoseprogramm vorgestellt, die in Größe und Anzahl ihrer Funktionen variieren (siehe Tabelle 2). Die Speichernutzung liegt zwischen 80–98 %.

Größe der Funktion in Wörtern	Anzahl Funktionen für verschiedene Programme						
	sf80	sf92	lf80	lf90	sn90	sn95	sn98
0–19	109	126	0	0	120	112	108
20–39	183	210	44	48	70	73	77
40–59	18	21	0	0	20	24	21
60–79	0	0	20	24	10	10	13
80–99	0	0	0	0	8	8	8
100–119	0	0	0	0	6	4	2
120–139	0	0	10	12	3	4	6
140–159	0	0	0	0	1	3	2
160–179	0	0	0	0	2	1	1
180–199	0	0	0	0	1	1	2
270–289	0	0	5	6	0	1	1
470–489	0	0	3	3	0	0	0
Anzahl der Funktionen	310	357	82	93	241	241	241
Speichernutzung (Bytes)	6565	7545	6560	7440	7440	7800	8028
Speichernutzung	80 %	92 %	80 %	90 %	90 %	95 %	98 %

Tab. 2: Verschiedene Diagnoseprogramme und deren Aufbau

Die Diagnoseprogramme sf* und lf* wurden künstlich erzeugt und sind aus kleinen bzw. großen Funktionen aufgebaut. Die Funktionen sind darüber hinaus zufällig angeordnet. Die Diagnoseprogramme sn* basieren auf einer echten Anwendung eines Sensorknotens mit einem MSP430-Prozessor. Die Größen der Funktionen wurden dafür geringfügig verändert, um verschiedene Speicherbereiche zu belegen. Die Funktionen wurden außerdem vom Linker in absteigender Größe sortiert.

Für die oben beschriebenen fehlerhaften Speicher werden nun die Diagnoseprogramme mit Hilfe der folgenden Methoden programmiert:

- Methode 1 a: Neuordnung der Funktionen
- Methode 1 b: Neuplatzierung der Funktionen in der gleichen Reihenfolge
- Methode 2: Zerteilung von Funktionen

Für die Neuordnung der Funktionen wurde eine einfache Heuristik verwendet, welche die größte passende Funktion dem nächsten fehlerfreien Speicherbereich zuweist. In Tabelle 3 ist dargestellt, wie viele Anwendungen erfolgreich angepasst werden konnten und welche neue Produktionsausbeute sich insgesamt daraus ergibt.

Die größte Ausbeute wird mit Methode 2 erzielt. Hier wird vor jedem fehlerhaften Speicherwort eine Branch-Operation eingefügt. Folglich muss der Reservespeicher höchstens doppelt so groß sein, wie die fehlerhaften Bereiche. Andererseits müssen auch Sprünge innerhalb der Funktionen angepasst werden. Außerdem wird die Laufzeit der Funktionen verlängert, was bei den anderen Fehlerbehandlungsmethoden nicht der Fall ist. Daher

	Methode 1 a			Methode 1 b			Methode 2		
	Ok	Fehler	Ausbeute	Ok	Fehler	Ausbeute	Ok	Fehler	Ausbeute
sf80	17	2	93,0 %	15	4	92,2 %	19	0	93,8 %
sf92	16	3	92,6 %	15	4	92,2 %	17	2	93,0 %
lf80	15	4	92,2 %	15	4	92,2 %	19	0	93,8 %
lf90	15	4	92,2 %	15	4	92,2 %	18	1	93,4 %
sn90	15	4	92,2 %	15	4	92,2 %	18	1	93,4 %
sn95	15	4	92,2 %	14	5	91,8 %	16	3	92,6 %
sn98	15	4	92,2 %	11	8	90,5 %	14	5	91,4 %

Tab. 3: Produktionsausbeute insgesamt bei 8K-Speichern für verschiedene Fehlerbehandlungsmethoden und Anwendungen

ist für sn98 Methode 2 schlechter als Methode 1 a. Insgesamt wird selbst mit der sehr einfachen Methode 1 b eine Ausbeute von über 90 % erzielt und damit um mehr als 4 % verbessert.

Ohne Neuordnung der Funktionen sinkt die Ausbeute für Anwendungen mit hohem Speicherverbrauch. Mit Neuordnung ist die Ausbeute quasi unabhängig vom verfügbaren Reservespeicher.

	Methode 1 a			Methode 1 b			Methode 2		
	Ok	Fehler	Ausbeute	Ok	Fehler	Ausbeute	Ok	Fehler	Ausbeute
sf80	33	0	93,4 %	33	0	93,4 %	33	0	93,4 %
sf92	33	0	93,4 %	33	0	93,4 %	33	0	93,4 %
lf80	33	0	93,4 %	33	0	93,4 %	33	0	93,4 %
lf91	33	0	93,4 %	33	0	93,4 %	33	0	93,4 %
sn90	33	0	93,4 %	33	0	93,4 %	33	0	93,4 %
sn95	33	0	93,4 %	33	0	93,4 %	33	0	93,4 %
sn98	33	0	93,4 %	31	2	92,6 %	33	0	93,4 %

Tab. 4: Produktionsausbeute insgesamt bei 16K-Speichern für verschiedene Fehlerbehandlungsmethoden und Anwendungen

Die gleiche Auswertung wurde ebenfalls für die 16K-Speicher durchgeführt. Die Anzahl der Funktionen der einzelnen Diagnoseprogrammen in Tabelle 2 wurde dazu verdoppelt. Die Ergebnisse sind in Tabelle 4 zusammengefasst. Methode 1 a und 2 können alle Defekte reparieren. Selbst die einfachste Methode 1 b kann fast alle Fehler behandeln; außer bei der Anwendung mit 98 % Speichernutzung.

In Tabelle 5 ist die erreichbare Verbesserung der Produktionsausbeute für 8K- und 16K-Bausteine zusammengefasst.

Für 8K-Bausteine kann die Ausbeute, je nach Speicherauslastung der Anwendung und der eingesetzten Reparaturmethode, von 86 % auf etwa 92 % erhöht werden, bei den 16K-Bausteinen sogar von 80 % auf etwa 93 %. Dabei wird die Ausbeute verbessert ohne zusätzliche Reservespeicher oder einer Modifikation des Speichercontroller. Folglich kann es günstig sein, die vorgestellten Methoden für einfache eingebettete Speicher anzuwen-

Speicherart	Anzahl	Ausbeute			
		ohne Reparatur	Methode 1 a	Methode 1 b	Methode 2
8K	244	86,1 %	92–93 %	90–92 %	91–94 %
16K	244	79,9 %	93 %	92–93 %	93 %

Tab. 5: Verbesserung der Produktionsausbeute für 8K- und 16K-Flash-Speicher mit softwarebasierter Reparatur

den, bei denen diese Hardwareerweiterungen aus Kostengründen nicht möglich sind. Es ist jedoch ein weiterer Schritt im Produktionsprozess erforderlich, der die Anpassung der Anwendung an Hand der gewählten Reparaturmethode durchführt.

5 Zusammenfassung

Es wurden verschiedene softwarebasierte Methoden vorgestellt, um permanente Fehler in Daten- und Programmspeichern in kleinen eingebetteten System zu Handhaben. Basierend auf detaillierten Diagnosedaten von produzierten Speicherbausteinen konnte gezeigt werden, dass softwarebasierte Methoden es ermöglichen mehr als 50 % der Fehler zu reparieren. Dabei kann im Gegensatz zu den hardwarebasierten Methoden ohne zusätzliche Hardware ausgekommen werden, indem die Redundanz in Form von unbelegten Speicherwörtern ausgenützt wird. Die vorgestellten Methoden eignen sich daher sehr gut die Produktionsausbeute von kleinen eingebetteten System zu Verbessern, die aus einfachen und kostengünstigen Komponenten aufgebaut sind, wie z. B. Flash-Speicher ohne Selbstreparatur.

Literaturverzeichnis

[AIB10] Ankolekar, P.; Isaac, R.; Bredow, J.: Multibit error-correction methods for latency-constrained flash memory systems. IEEE Transactions on Device and Materials Reliability, 10(1):6, 2010.

[CBT09] Cao, Yu; Bose, Pradip; Tschanz, Jim: Reliability Changes in Nano-CMOS Design (Guest Editor's Introduction). IEEE Design & Test of Computers, 26(6):6–7, 2009.

[CG11] Cheng, Da; Gupta, Sandeep: A novel software-based defect-tolerance approach for application-specific embedded systems. In: IEEE 29th International Conference on Computer Design (ICCD'11). S. 443–444, 2011.

[Ch12] Chu, Chia-Ching; Lin, Yi-Min; Yang, Chi-Heng; Chang, Hsie-Chia: , A Fully Parallel BCH Codec with Double Error Correcting Capability for NOR Flash Applications, 2012.

[Go03] Goloubeva, O.; Rebaudengo, M.; Reorda, M.S.; Violante, M.: Soft-error detection using control flow assertions. In: Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on. S. 581–588, Nov 2003.

[HCW06] Hsiao, Yu-Ying; Chen, Chao-Hsun; Wu, Cheng-Wen: , A built-in self-repair scheme for NOR-type flash memory, 2006.

[HCW07] Huang, Rei-Fu; Chen, Chao-Hsun; Wu, Cheng-Wen: Economic Aspects of Memory Built-in Self-Repair. IEEE Design & Test of Computers, 24(2):8, 2007.

[HCW10] Hsiao, Yu-Ying; Chen, Chao-Hsun; Wu, Cheng-Wen: Built-In Self-Repair Schemes for Flash Memories. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 29(8):13, 2010.

[Li03] Li, Jin-Fu; Yeh, Jen-Chieh; Huang, Rei-Fu; Wu, Cheng-Wen: A built-in self-repair scheme for semiconductor memories with 2-D redundancy. In: Test Conference, 2003. Proceedings. ITC 2003. International. Jgg. 1, S. 393–402, 2003.

[Li06] Lisboa, C.A.L.; Carro, L.; Reorda, M.S.; Violante, M.: Online hardening of programs against SEUs and SETs. In: Defect and Fault Tolerance in VLSI Systems, 2006. DFT '06. 21st IEEE International Symposium on. S. 280–290, Oct 2006.

[MSV13] Müller, Sebastian; Schölzel, Mario; Vierhaus, Heinrich Theodor: Towards a Graceful Degradable Multicore-System by Hierarchical Handling of Hard Errors. In: Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on. S. 302–309, Feb 2013.

[OSM00] Oh, N.; Shirvani, P.P.; McCluskey, E.J.: Control-Flow Checking by Software Signatures. IEEE Transactions on Reliability, 51(2):111–122, 2000.

[Pe08] Pekmestzi, K.; Axelos, N.; Sideris, I.; Moshopoulos, N.: , A BISR Architecture for Embedded Memories, 2008.

[Re99] Rebaudengo, M.; Reorda, M.Sonza; Torchiano, M.; Violante, M.: Soft-error Detection through Software Fault-Tolerance techniques. In: Proc. of the 14th International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT'99). Proc. of the 14th International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT'99), S. 210–218, 1999.

[Sc10] Schölzel, Mario: Software-Based Self-Repair of Statically Scheduled Superscalar Data Paths. In: Proc. of the 13th IEEE International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS'10). Proc. of the 13th IEEE International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS'10), S. 66–71, 2010.

[Sc11] Schölzel, Mario: Fine-Grained Software-Based Self-Repair of VLIW Processors. In: 26th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems. 26th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, S. 41–49, 2011.

[SS11] Skoncej, Patryk; Schoof, Gunther: , Getting ready for non-volatile memories, 2011.