Hardware-Software-Co-Synthese zur Verbesserung der Fehlertoleranz

Stefan Frehse[1] Heinz Riener[1] Görschwin Fey[1,2]

[1]Fachbereich 3 - Mathematik und Informatik
Universität Bremen
{sfrehse,hriener,fey}@informatik.uni-bremen.de

[2]Institut für Raumfahrtsysteme
DLR Bremen
goerschwin.fey@dlr.de

ZUSAMMENFASSUNG

Eingebettete Systeme bestehen aus Hardware und Software und werden in sicherheitskritischen Bereichen wie z.B. in der Luft- und Raumfahrt eingesetzt. Durch die zunehmende Integrationsdichte moderner, digitaler Schaltkreise steigt die Anfälligkeit dieser Systeme für transiente Fehler. Techniken um eingebettete Systeme gegen transiente Fehler abzusichern, d.h. die *Fehlertoleranz* dieser Systeme zu steigern, werden oft entweder nur in Hardware oder nur in Software implementiert.
In dieser Arbeit befassen wir uns mit der Synthese von Techniken zur Steigerung der Fehlertoleranz eines eingebetteten Systems unter Berücksichtigung von Hardware und Software. Wir nutzen einen neuartigen Ansatz basierend auf Modellprüfung um die Fehlertoleranz von Softwareprogrammen unter Betrachtung des Maschinencodes zu bewerten. In diesen Ansatz wird ein existierendes Bewertungsverfahren für Hardwarekomponenten eingebettet. Weiter präsentieren wir einen iterativen Algorithmus, der ein eingebettetes System mit unserem Verfahren bewertet und Techniken zur Fehlertoleranz in Hardware und Software synthetisiert. Wir evaluieren den Algorithmus in einer Fallstudie für ein eingebettetes System zur Kollisionsvermeidung von Flugzeugen.

ABSTRACT

Embedded systems consist of hardware and software and are ubiquitous in safety critical fields, e.g., aerospace. The increasing integration density of modern, digital circuits causes an increasing vulnerability of embedded systems to transient faults. Techniques to protect embedded systems against transient faults, i.e., to increase the *fault tolerance* of the systems, are often either implemented only in hardware or only in software.
In this paper, we focus on the synthesis of techniques to improve the fault tolerance of embedded systems considering hardware and software. We use a new approach based on model checking to assess the fault tolerance of software programs utilizing their machine code. In this approach, we embed an existing method for assessing fault tolerance for hardware. Moreover, we present an iterative algorithm for assessing the fault tolerance of an embedded system leveraging our approach and for synthesizing technique to improve the fault tolerant in hardware and software. We evaluate the algorithm in a case study using an embedded system which instructs aircraft to avoid collisions.

I. EINFÜHRUNG

Eingebettete Systeme bestehen sowohl aus Hardware als auch aus Software. In sicherheitskritischen Anwendungen, wie z.B. in der Luft- und Raumfahrt, ist die Zuverlässigkeit dieser Systeme von besonderer Bedeutung. Wir betrachten *transiente Fehler*. Beispielsweise kann die Energie einfallender „kosmischer Strahlung“ den Signalpegel an einer Signalleitung kurzzeitig erhöhen und den Zustand des Systems beeinflussen.

Die Anfälligkeit eingebetteter Systeme gegenüber transienten Fehlern steigt mit zunehmender Integrationsdichte digitaler Schaltkreise [3]. Neben der Abschirmung eines Systems gegen Strahlung, helfen Techniken zur Fehlertoleranz in Hardware und Software, Auswirkungen transienter Fehler zu erkennen und zu korrigieren.

Jede Technik zielt auf die redundante Berechnung kritischer Daten ab und wirkt sich auf die Kosten des eingebetteten Systems aus [2]. Als Kostenmaße dienen beispielsweise in Hardware die benötigte Chipfläche oder die Leistungsaufnahme eines Schaltkreises und in Software die Ausführungszeit oder die Anzahl der benötigten Instruktionen. Wie verschiedene Techniken kombiniert werden müssen, um eine möglichst hohe Fehlertoleranz unter Einhaltung vorgegebener Kosten zu erreichen, ist im Allgemeinen nicht trivial entscheidbar.

In dieser Arbeit befassen wir uns mit Methoden zur Bewertung und zur Verbesserung der Fehlertoleranz eingebetteter Systeme. Existierende Verfahren beschränken ihre Betrachtun-

Diese Arbeit wurde durch die Deutsche Forschungsgemeinschaft unter dem Kennzeichen FE 797/6-1 unterstützt.

gen entweder nur auf Hardware [13], [9], [4], [7], nur auf Software [15], [17], oder operieren auf abstrakten Modellen [10]. Wir verwenden formale Methoden und berücksichtigen sowohl Hardware als auch Software.

Wir nutzen einen neuartigen Ansatz, der die Fehlertoleranz von Software unter Verwendung des Maschinencodes bewertet: Eine Instruktion ist *robust*, wenn ein in der Hardware auftretender, transienter Fehler in Software erkannt oder korrigiert wird, und *nicht-robust*, wenn sich ein transienter Fehler auf die Ausgaben des Programms auswirken kann. Für nicht-robuste Instruktionen wird die Berechnung der Fehlertoleranz präzisiert indem die Hardware berücksichtig wird, die die Instruktion ausführt.

Weiter präsentieren wir einen Algorithmus, der die Fehlertoleranz eines eingebetteten Systems unter Einhaltung einer gegebenen Kostenschranke maximiert. In einem iterativen Prozess werden Teile der Software und der Hardware nach einer Heuristik ausgewählt und gehärtet, solange die gegebene Kostenschranke nicht überschritten wird.

Wir evaluieren den Algorithmus in einer Fallstudie für ein eingebettetes System zur Kollisionsvermeidung von Flugzeugen. Als Software dient das ANSI-C Programm *Traffic Collision Avoidance System* (TCAS) aus dem *Software-Artifact Infrastructure Repository* (SIR) [6]. Als Hardware dient eine fiktive CPU mit einem RISC-ähnlichen Befehlssatz. Die Hardwarekomponenten der CPU stehen jeweils in unterschiedlichen Realisierungen zur Verfügung und unterscheiden sich in ihrer Anfälligkeit für transiente Fehler und ihren Kosten. Unser Algorithmus synthetisiert ein System mit einer Fehlertoleranz von 100% bei gleichzeitiger Härtung in Hardware und Software. Die Kosten sind deutlich geringer als die Kosten einer Härtung, die nur in Hardware oder nur in Software realisiert werden würde.

Die Gliederung dieser Arbeit ist wie folgt: In Abschnitt II präsentieren wir Grundlagen. In Abschnitt III beschreiben wir die Berechnung der Fehlertoleranz für Hardware und Software. In Abschnitt IV führen wir den Algorithmus zur Verbesserung der Fehlertoleranz ein. In Abschnitt V evaluieren wir den Algorithmus in einer Fallstudie und diskutieren experimentelle Ergebnisse. Abschnitt VI fasst diese Arbeit zusammen.

II. Grundlagen

Hardware: Wir betrachten eine CPU C, auf der ein Programm P in Maschinencode ausgeführt wird. Die CPU ist als eine Menge $C = \{c_1, c_2, \ldots, c_n\}$ von Hardwarekomponenten gegeben. Jede Hardwarekomponente c besitzt einen Typ $\mathsf{typ}(c) \in \mathsf{Ops}$, wobei $\mathsf{Ops} = \{\texttt{MULT}, \texttt{ADD}, \texttt{DIV}, \ldots\}$ für die Menge der von der CPU unterstützten Operationen steht. Weiter existieren zu jeder Hardwarekomponente c eine oder mehrere Realisierungen $\mathsf{alt}(c) = \{c^{(1)}, c^{(2)}, \ldots, c^{(k)}\}$, die die Funktionalität der Hardwarekomponente als unterschiedliche Schaltkreise realisieren. Hierdurch lässt sich beispielsweise eine ALU bestehend aus verschiedenen Teilschaltkreisen, die arithmetische und logische Operationen realisieren, zerlegen, um sie später optimal zu konfigurieren. Das bedeutet, wenn die Abbildung der Zerlegungen entsprechend gewählt wird, berechnet der neue dieser Arbeit Algorithmus welche Operationen der ALU robust ausgelegt werden müssen.

Wir beschränken uns auf die Betrachtung von kombinatorischen Schaltkreisen. Eine Realisierung $c^{(j)} \in \mathsf{alt}(c)$ wird mit zwei Maßen bewertet: (1) Die *Fehlertoleranz* $\mathsf{ft}_{\mathsf{HW}}(c^{(j)}) \in \mathbb{R}$ ist ein Maß für die Fähigkeit der Realisierung unter Einwirkung transienter Fehler korrekte Berechnungen auszuführen und kann unter Einsatz von formalen Methoden wie z.B. [7], [13] bestimmt werden. (2) Die *Kosten* $\mathsf{kosten}_{\mathsf{HW}}(c^{(j)}) \in \mathbb{R}$ spiegeln den Aufwand der Realisierung z.B. in der Anzahl der benötigten Gatter oder der Leistungsaufnahme des Schaltkreises wider.

Software: Ein Programm P ist eine Sequenz von Instruktionen $i \in \mathsf{Inst}^*$ mit einer Menge von Eingaberegistern und einer Menge von Ausgaberegistern, wobei Inst^* für die Menge alle Instruktionen steht. Die Menge $\mathsf{Inst} \subseteq \mathsf{Inst}^*$ ist die Menge der Instruktionen, die in der vorliegende Arbeit als nicht-robust bzw. robust klassifiziert werden. Diese Instruktionen können beispielsweise arithmetische oder logische Operationen sein. Ausgenommen sind beispielsweise Load/Store Instruktionen oder Instruktionen die den Kontrollfluss beeinflussen. In dieser Arbeit betrachten wir transformative Programme, die stets terminieren und ein Ergebnis in den Ausgaberegistern abspeichern. Wir definieren den längsten Pfad eines Programms als die längste Sequenz von Instruktionen vom Auslesen eines Eingaberegisters bis zum Abspeichern eines Ergebnisses in einem Ausgaberegister.

Jede Instruktion $i \in \mathsf{Inst}$ besitzt einen Typ $\mathsf{typ}(i) \in \mathsf{Ops}$ und kann von einer Realisierung einer Hardwarekomponente desselben Typs ausgeführt werden, d.h., Instruktion i kann von einer Realisierung $c^{(j)} \in alt(c)$ Hardwarekomponente c ausgeführt werden *genau dann wenn* (gdw) $\mathsf{typ}(i) = \mathsf{typ}(c^{(j)})$. Beispielsweise kann eine Instruktion mit Typ `MULT` von einer beliebigen Realisierung eines Multiplizierers der CPU ausgeführt werden.

III. Berechnung der Fehlertoleranz

Eingebettete Systeme bestehen aus Hardware und Software. Wir verwenden ein zweistufiges Verfahren zur Berechnung der Fehlertoleranz, das sowohl die Hardware als auch die Software berücksichtigt. In der ersten Stufe wird die Fehlertoleranz der Software bestimmt, d.h. die Auswirkungen transienter Fehler auf einzelne Instruktionen im Maschinencode werden berechnet. Sobald sich ein Fehler in einer Instruktion auf ein Ausgaberegister des Programms auswirkt, wird die betreffende Instruktion als nicht-robust klassifiziert, unabhängig davon auf welcher Hardwarekomponente die Instruktion ausgeführt wird.

Treten Fehler innerhalb des Programmzählers der CPU auf, kann eine sinnvolle Ausführung eines Programms trotz

Techniken zur Fehlertoleranz in Software nicht gewährleistet werden. Techniken, um den Programmzähler fehlertolerant zu implementieren, sind beispielsweise in der Arbeit [1] beschrieben. Aus diesem Grund werden Instruktionen, die den Programmzähler manipulieren, als robust angenommen und nicht weiter berücksichtigt.

Die Berechnung ist konservativ, da die Hardware vorerst als nicht-robust angenommen wird, unabhängig davon, welche Techniken zur Fehlertoleranz tatsächlich in Hardware implementiert sind. In der zweiten Stufe wird die Fehlertoleranz der Hardware berücksichtigt. Die Fehlertoleranz einer nicht-robusten Instruktion wird durch die Fehlertoleranz der Realisierung in Hardware, auf der die Instruktion ausgeführt wird, ersetzt.

A. Software-Fehlertoleranz

Für die Berechnung der Fehlertoleranz in Software verwenden wir Modellprüfung. Unser Ansatz ist eine Kombination aus SAT-basierter Äquivalenzprüfung [11] für Software und Robustheitsprüfung für Hardware [7]. Wir konstruieren eine logische Formel, die erfüllbar ist gdw sich mindestens ein in der Hardware auftretender, transienter Fehler auf mindestens ein Ausgaberegister des Programms auswirken kann. Ein auftretender Fehler in Hardware wird als nicht-deterministisches Ergebnis einer Instruktion in Software modelliert. Dieses Modell berücksichtigt nicht nur ein einzelnes gekipptes Bit im Ergebnis, sondern *lokale Mehrfachfehler* bei denen sogar alle Bits abweichen können. Die Semantik des Programms wird hierzu zweifach kodiert. Beide Kodierungen verwenden dieselben Variablen für die Eingaberegister. Beim ersten Kodiervorgang bleibt die Semantik des Programms unverändert. Beim zweiten Kodiervorgang wird für jede Instruktion zusätzliche Fehlerlogik hinzugefügt um einen nicht-deterministischen Wert zu injizieren. Die Fehlerlogik kann verwendet werden, um die Semantik einer Instruktionen aus der logischen Formel zu entkoppeln, d.h. das Ergebnis der Instruktion entspricht dann einer offenen Variable.

Wir verwenden einen *Satisfiability Modulo Theories* (SMT)-Beweiser, um die Erfüllbarkeit der logischen Formel zu überprüfen. Der Beweiser sucht nach erfüllenden Belegungen, sodass sich die Augaberegister unterscheiden, wenn die Fehlerlogik höchstens eine Instruktion entkoppelt. Jede erfüllende Belegung beweist, dass die jeweilige entkoppelte Instruktion in Software nicht-robust ist. Wir verwenden den Beweiser um alle nicht-robusten Instruktionen iterative aufzuzählen. Aus jeder erfüllenden Belegung extrahieren wir die entkoppelte Instruktion und beschränken die Wahlmöglichkeiten der Fehlerlogik, sodass keine Instruktion öfter als einmal entkoppelt werden kann. Wenn die logische Formel unerfüllbar ist, sind alle übrigen Instruktionen als robust nachgewiesen, da sich kein Fehler auf die Ausgaberegister auswirken kann.

Die Fehlertoleranz $\mathsf{ft}_{\mathrm{SW}}$ in Software eines Programms P berechnet sich als Verhältnis der Anzahl der nicht-robusten Instruktionen zur Anzahl aller Instruktionen. Sei P ein Programm mit Instruktionen Inst und $\mathcal{S} \subseteq \mathsf{Inst}$ die Menge der nicht-robusten Instruktionen, dann ist $\mathsf{ft}_{\mathrm{SW}}$ wie folgt definiert:

$$\mathsf{ft}_{\mathrm{SW}}(\mathsf{P}) = \frac{|\mathsf{Inst} \setminus \mathcal{S}|}{|\mathsf{Inst}|} \tag{1}$$

B. Einbettung der Hardware-Fehlertoleranz

Instruktionen, die in Software als robust nachgewiesen sind, können auf eine beliebige Realisierung einer Hardwarekomponente des jeweiligen Typs abgebildet werden, da die Instruktionen bereits in Software gegen transiente Fehler abgesichert sind.

Für Instruktionen, die in Software als nicht-robust klassifiziert wurden, berücksichtigen wir die Hardware, auf der die Instruktion ausgeführt wird. Wir bewerten die Fehlertoleranz $r(i, \Phi)$ einer Instruktion $i \in S \subseteq \mathsf{Inst}$, wie folgt

$$r(i, \Phi) = \begin{cases} \mathsf{ft}_{\mathrm{HW}}(\Phi(i)) & \text{wenn } i \in \mathcal{S} \subseteq \mathsf{Inst} \\ 1 & \text{sonst,} \end{cases} \tag{2}$$

wobei die Abbildung Φ eine Instruktion i zu der Realisierung einer Hardwarekomponente zuordnet, auf der sie ausgeführt wird.

Um die gesamte Fehlertoleranz ft eines eingebetteten Systems bestehend aus Hardware und Software zu bestimmen, wird die Fehlertoleranz jeder Instruktion des Programms P unter Berücksichtigung der Abbildung Φ der Instruktionen auf Hardware aufsummiert und durch die Anzahl der Instruktionen geteilt:

$$\mathsf{ft}(\mathsf{P}, \Phi) = \frac{\sum\limits_{i \in \mathsf{Inst}} r(i, \Phi)}{|\mathsf{Inst}|} \tag{3}$$

C. Kostenmodell

Jede Härtung, ob in Hardware oder in Software realisiert, verursacht zusätzliche Kosten und beeinflusst Eigenschaften, wie z.B. Laufzeit und Leistungsaufnahme, des eingebetteten Systems. Um eine optimale Kombination verschiedener Techniken hinsichtlich der Fehlertoleranz unter Einhaltung gegebener Kosten zu gewährleisten, wird ein Kostenmodell vorgegeben.

Das Kostenmodell berücksichtigt *Platzkosten* $\mathsf{kosten}_{\mathrm{P}}$ und *Laufzeitkosten* $\mathsf{kosten}_{\mathrm{L}}$. Die Platzkosten sind definiert als

$$\mathsf{kosten}_{\mathrm{P}}(\mathsf{P}, \Phi) = \sum_{c \in H} \mathsf{kosten}_{\mathrm{HW}}(c) \text{ mit } H = \bigcup_{i \in \mathsf{Inst}} \Phi(i).$$

Sie entsprechen der Summe der benötigten Platzkosten aller Realisierungen auf denen Instruktionen ausgeführt werden.

Die Laufzeitkosten ergeben sich durch die Berechnung des längsten Ausführungspfades im betrachteten Programms P:

$$\mathsf{kosten}_{\mathrm{L}}(\mathsf{P}) = \mathsf{längsterPfad}(\mathsf{P}).$$

Da alle Realisierungen in Hardware durch kombinatorische Schaltkreise implementiert werden, nehmen wir die Laufzeiten für alle Typen von Instruktionen konstant als 1 an.

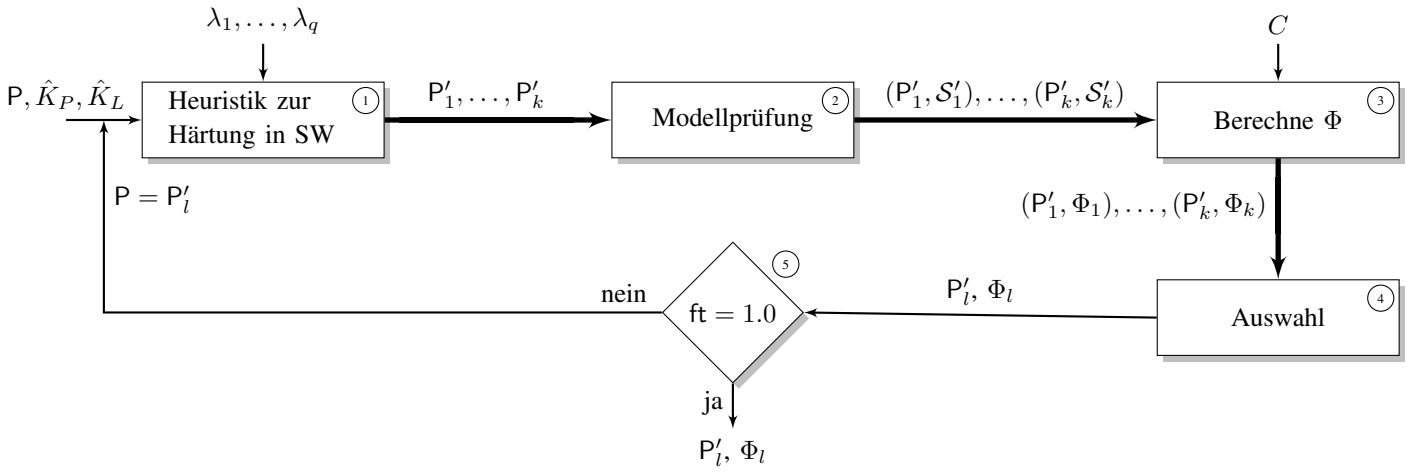

Abbildung 1. Iterativer Prozess zur Verbesserung der Fehlertoleranz

Für die vorliegende Arbeit wurde das Kostenmodell einfach gewählt. Das Kostenmodell kann an andere Bedingungen geknüpft und erweitert werden.

IV. Synthese von Techniken zur Fehlertoleranz

In diesem Abschnitt stellen wir einen iterativen Algorithmus vor, der die Fehlertoleranz eines eingebetteten Systems bewertet und Techniken zur Verbesserung der Fehlertoleranz in Hardware und Software berücksichtigt.

In Abbildung 1 ist der Ablauf des Algorithmus schematisch dargestellt. Jede Iteration des Algorithmus ist in fünf Schritte unterteilt. Als Eingabe dient ein Programm P, die Menge C der zugrundeliegenden Hardwarekomponenten und die maximalen Platzkosten $\hat{K}_P$ und maximalen Laufzeitkosten $\hat{K}_L$. Die maximalen Platzkosten beschränken den Platzbedarf der Hardware. Die maximalen Laufzeitkosten beschränken den längsten Ausführungspfad in einem Softwareprogramm. Zusätzlich nimmt die Heuristik zur Härtung der Software Parameter $\lambda_1, \ldots, \lambda_q$ entgegen. Diese Parameter unterscheiden sich je nach gewählter Heuristik. Die Ausgaben des Algorithmus in der l-ten Iteration sind ein Programm P'_l, das funktional äquivalent zu dem Programm P ist, und eine Abbildung Φ_l von Instruktionen zu Realisierungen. Die Abbildung Φ_l legt fest, welche Instruktion auf welcher Realisierung in Hardware ausgeführt werden soll.

A. Schritt 1: Heuristik zur Härtung in Software

Im ersten Schritt wendet der Algorithmus Techniken zur Verbesserung der Fehlertoleranz auf das Programm P an. Eine Verbesserung der Fehlertoleranz kann in Software durch existierende Techniken [16], [15], [14], [18], [17] erzielt werden, die den Maschinencode instrumentieren, um Fehler zu erkennen oder zu korrigieren. Eine erschöpfende Exploration der Möglichkeiten existierende Techniken in Software anzuwenden und zu kombinieren, ist nur unter großem Aufwand möglich. Deshalb verwenden wir eine Heuristik um zu entscheiden, welche Technik auf welche Instruktionen angewandt wird. Als Ergebnis dieses Schrittes werden k verschieden gehärtete Programme $\mathsf{P}'_1, \ldots, \mathsf{P}'_k$ erzeugt, die funktional äquivalent zum Programm P sind. Gehärtete Programme, deren Laufzeitkosten $\mathsf{kosten}_{\mathrm{L}}(\mathsf{P}'_l)$ die vorgegebene Kostenschranke $\hat{K}_L$ überschreiten, werden verworfen. Wenn kein Programm die Kostenschranke einhält, terminiert der Algorithmus frühzeitig mit dem Ergebnis der vorherigen Iterationen.

B. Schritt 2: Modellprüfung

Im zweiten Schritt verwenden wir einen Modellprüfer, um die Fehlertoleranz der gehärteten Programme zu bestimmen. Unser Modellprüfer operiert auf *Low Level Virtual Machine* (LLVM) [12], einer RISC-ähnlichen Assemblersprache. Rekursionen und Schleifen werden gegebenenfalls abgerollt [11]. Der Modellprüfer verwendet das Verfahren aus Abschnitt III-A und berechnet für jedes gehärtete Programm P_l die Menge $\mathcal{S}_l$ der nicht-robusten Instruktionen unter Verwendung eines SMT-Beweisers. Als Schnittstelle zum Beweiser dient eine domänenspezifische Sprache bereitgestellt durch metaSMT [8].

C. Schritt 3: Zuordnung der Instruktionen zu Realisierung

Im dritten Schritt ordnen wir die Instruktionen zu Realisierungen in Hardware zu, auf denen die Instruktionen ausgeführt werden. Um die Fehlertoleranz des eingebetteten Systems zu maximieren, müssen die nicht-robusten Instruktionen $i \in S_l$ möglichst von Realisierungen ausgeführt werden, die in Hardware gegen transiente Fehler gehärtet wurden. Für jedes Programm P'_l wird die optimale Abbildung Φ_l bestimmt, die unter Einhaltung der Platzkosten $\mathsf{kosten}_{\mathsf{P}}(\mathsf{P}'_l, \Phi_l) < \hat{K}_P$ die höchste Fehlertoleranz des eingebetteten Systems gewährleistet.

Zur Bestimmung der optimalen Abbildung formulieren wir ein Optimierungsproblem, in dem alle Zuordungsmöglichkeiten als Bedingungen modelliert werden. Das Optimierungsproblem

wird in eine logische Formel mit Pseudo-Booleschen Operatoren abgebildet. Wir lösen das Optimierungsproblem mit einem SMT-Beweiser, indem wir nach erfüllenden Belegungen suchen und systematisch Bedingungen zur logischen Formel hinzufügen bis ein Optimum gefunden wird. Das Optimum entspricht einer Abbildung Φ_l, die die höchste Fehlertoleranz unter Einhaltung der gegeben Kostenschranke erreicht.

Wenn die logische Formel unerfüllbar ist, gibt es für das jeweilige Programm keine Abbildung, die die vorgegebene Kostenschranke nicht überschreitet. Das entsprechende Programm wird dann verworfen. Wenn kein Programm die Kostenschranke einhält, terminiert der Algorithmus frühzeitig mit dem Ergebnis der vorherigen Iterationen.

D. Schritt 4: Auswahl von Programm und Zuordnung

Im vierten Schritt wird aus den Paaren von Programmen und Abbildungen $(P'_1, \Phi_1), \ldots, (P'_k, \Phi_k)$ das Programm P_l hinsichtlich der Fehlertoleranz unter Berücksichtigung der Zuordnung Φ_l zu Realisierungen in Hardware ausgewählt, welches die höchste Fehlertoleranz $\mathsf{ft}(P'_l, \Phi_l)$ aufweist.

E. Schritt 5: Terminierungskriterium

Im fünften Schritt überprüfen wir, ob eine maximale Fehlertoleranz von 100% erreicht wurde. Wenn ja, terminiert der Algorithmus mit der Ausgabe des Programms P'_l und einer Zuordnung zu Realisierung Φ_l. Anderenfalls wird eine neue Iteration mit Programm P_l als Eingabe gestartet.

V. Fallstudie

Wir evaluieren den Algorithmus in einer Fallstudie für ein eingebettetes System bestehend aus Hardware und Software. Als Software wurde das Programm TCAS zur Kollisionsvermeidung von Flugzeugen verwendet. Die zugrundeliegende Hardware entspricht einer fiktiven CPU. Für jeden vorkommenden Typ einer Instruktion im Programm TCAS, wurde ein Schaltkreis, d.h. eine Realisierung in Hardware, modelliert. Um die Schaltkreise gegen transiente Fehler zu schützen, wird *Triple Modular Redundancy* (TMR) auf alle Schaltkreise angewandt. Demnach existieren zwei Realisierungen in Hardware für jeden Typ einer Instruktion. Insgesamt werden 14 Schaltkreise modelliert. Als Platzkosten wird die Anzahl der Gatter von allen eingesetzten Realisierungen in Hardware bestimmt. Der Algorithmus erlaubt den Einsatz verschiedener Heuristiken zur Härtung von Software. In dieser Arbeit beschränken wir uns auf eine Heuristik basierend auf der Technik *Error Detection by Duplicated Instructions* (EDDI) [15]. Der Algorithmus wählt aus, welche Teile in Software mit EDDI und welche Realisierungen in Hardware mit TMR gehärtet werden, um unter Einhaltung vorgegebener Kosten eine Fehlertoleranz von 100% zu erreichen.

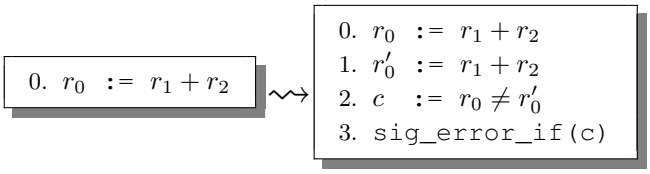

Abbildung 2. Härtung einer Instruktion mit EDDI

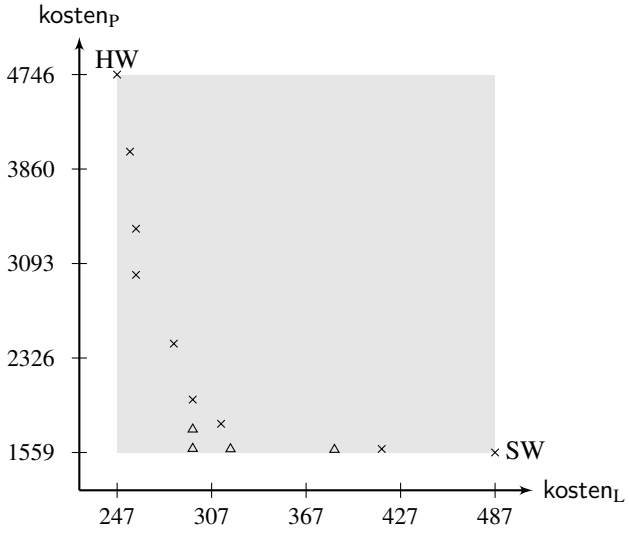

Abbildung 3. Entwurfsraum von TCAS

A. Heuristik basierend auf EDDI

Wir verwenden eine einfache Heuristik basierend auf EDDI. Zunächst werden die verschiedenen Typen von Instruktionen nach den Kosten für die robuste Realisierung in Hardware in einer Prioritätsliste absteigend geordnet. Die Heuristik wählt den ersten Typ aus der Prioritätsliste, härtet alle Instruktionen von diesem Typ und erzeugt ein neues gehärtetes Programm P'. Wenn P' die Laufzeitkosten nicht einhält, wird der Vorgang mit dem nächsten Typ der Prioritätsliste wiederholt. Anderenfalls wird das Programm zurückgegeben. Wurden alle Typen in der Prioritätsliste betrachtet, bricht der Algorithmus frühzeitig ab, und gibt das Ergebnis der letzten Iteration zurück.

Abbildung 2 zeigt wie eine einzelne Instruktion mit EDDI instrumentiert wird. Auf der linken Seite befinden sich der unveränderte Maschinencode, der zwei Register r_1 und r_2 addiert und im Register r_0 sichert. Auf der rechten Seite befindet sich der instrumentierte Maschinencode. Die Instruktion zur Addition wird dupliziert und ihr Ergebnis in einem weiteren Register r'_0 gesichert. Unterscheiden sich die Ergebnisse, der zweifach-redundant berechneten Addition, wird ein Fehler signaliert. Die Funktion `sig_error_if` ist ein nicht implementierter Funktionsrumpf, der dem Modellprüfer zur Identifikation der Fehlersignalisierung dient. Die Fehlersignalisierung kann im eingebetteten System beispielsweise als zusätzlicher Ausgang berücksichtigt werden, auf den die Hardware reagieren kann.

Tabelle I
KOSTEN UND FEHLERTOLERANZEN FÜR TCAS

Experiment	$\hat{K}_\mathrm{P}$	$\hat{K}_\mathrm{L}$	$\mathsf{kosten}_\mathrm{P}$	$\mathsf{kosten}_\mathrm{L}$	Fehlertoleranz	Härtung	Laufzeit	Iteration
SW	1559	∞	1559	487	100%	74	20,51s	1
HW	∞	247	4746	247	100%	0	15,91s	1
1	4000	247+50	3999	253	100%	1	23,44s	1
2	3500	247+50	3482	259	100%	2	32,76s	2
3	3000	247+50	2998	259	100%	2	34,99s	2
4	2500	247+50	2440	283	100%	6	202,80s	4
5	2000	247+50	1983	295	100%	10	90,58s	6
6	1800	247+50	1747	295	98%	13	82,13s	7
7	1800	247+100	1792	313	100%	15	86,64s	7
8	1600	247+50	1590	295	92%	13	22,98s	6
9	1600	247+100	1587	319	94%	21	24,59s	8
10	1600	247+150	1582	385	96%	41	24,21s	9
11	1600	247+200	1588	415	100%	50	31,02s	10

B. Experimentelle Ergebnisse

Die Experimente wurde auf einem Linux-Rechner mit einer Intel Core i7 CPU (2.4 GHz) und einem Hauptspeicher von 8 GB durchgeführt. Als SMT-Beweiser wurde Z3 [5] in der Version 3.2 verwendet. Der Speicherverbrauch lag bei allen Experimenten unter 150 MB.

Tabelle I listet experimentelle Ergebnisse für TCAS auf. Die Tabelle ist wie folgt aufgebaut: Die erste Spalte wird dazu verwendet um ein Experiment eindeutig zu identifizieren. Die zweite und dritte Spalte zeigen vorgegebene Kostenschranken für Platz- und Laufzeitkosten. Das Symbol ∞ in diesen Spalten bezeichnet unbeschränkte Kosten. Die vierte und fünfte Spalte geben die durch den Algorithmus berechneten Platz- und Laufzeitkosten an. Die sechste und siebte Spalte zeigen die erzielte Fehlertoleranz und die Anzahl, der in Software gehärteten Instruktionen. Die letzten beiden Spalten beschreiben die benötigte Laufzeit und die Anzahl der Iterationen des Algorithmus.

Die Experimente HW und SW beschreiben zwei Spezialfälle in denen die Härtung entweder nur in Hardware oder nur in Software erfolgt. Für den Fall HW wird das Programm ohne Instrumentierung verwendet. Alle Hardwarekomponenten werden auf eine Realisierung mit möglichst hoher Fehlertoleranz abgebildet. Für den Fall SW werden alle Instruktionen in Software mit EDDI gehärtet und auf eine Realisierung in Hardware mit möglichst geringen Kosten abgebildet.

In den Experimenten 1 bis 11 wurde der Algorithmus jeweils mit verschieden Kostenbeschränkungen mit der Heuristik aus Abschnitt V-A gestartet. Die maximalen Platzkosten wurden mit 4000, 3500, 3000, 2000, 1800, und 1600 gewählt. Die maximalen Laufzeitkosten wurden in 50iger Schritten relativ zum längsten Ausführungspfad des nicht instrumentierten Programms gewählt, d.h. $\hat{K}_L \in \{247, 247+50, \ldots, 247+200\}$. Die maximalen Laufzeitkosten wurden immer dann um 50 erhöht, wenn der Algorithmus keine Fehlertoleranz von 100% erreichen konnte. Beispielsweise bei Experiment 6 ist der Algorithmus frühzeitig terminiert, da keine Lösung gefunden wurde, bei der das Programm die Laufzeitkostenbeschränkung von 247+50 einhält.

Abbildung 3 zeigt ein Streudiagramm für die ermittelten Platz- und Laufzeitkosten für alle Experimente aus Tabelle I. Auf der horizontalen und der vertikalen Achse sind die Laufzeit- und Platzkosten aufgetragen. Ein Kreuz markiert die Kosten für ein Experiment, bei dem der Algorithmus mit einer Fehlertoleranz von 100% terminiert. Ein Dreieck markiert die Kosten für ein Experiment in dem der Algorithmus frühzeitig abbricht, da eine der vorgegebenen Kostenschranken nicht eingehalten werden kann.

Insgesamt wurden Lösungen mit ausgeglichenen Kosten innerhalb von kurzer Laufzeit gefunden. Wenn nur in Hardware bzw. Software gehärtet wird, sind die einzelnen Kosten sehr hoch. Die Härtung in Hardware verlangt 204% mehr Platzkosten um eine Fehlertoleranz von 100% zu erreichen. Die Härtung in Software verlangt 97% mehr Laufzeitkosten um eine Fehlertoleranz von 100% zu erreichen. Entwürfe mit ausgeglichenen Kosten können jedoch durch den neuen Algorithmus effektiv gefunden werden. Im Experiment 7 beispielsweise, kann eine Fehlertoleranz von 100% bei zusätzlichen 14% mehr Platzkosten und 26% mehr Laufzeitkosten erreicht werden.

VI. ZUSAMMENFASSUNG

In dieser Arbeit haben wir uns mit der Synthese von Techniken zur Steigerung der Fehlertoleranz eines eingebetteten Systems unter Berücksichtigung von Hardware und Software befasst. Wir haben einen neuartiges Verfahren zur Berechnung der Fehlertoleranz von Softwareprogrammen vorgestellt, das auf ein Bewertungsverfahren zurückgreift. Weiter haben wir einen Algorithmus zur Synthese von Techniken der Fehlertoleranz in Software eingeführt und in einer Fallstudie für ein eingebettetes System zur Kollisionvermeidung von Flugzeugen untersucht.

LITERATUR

[1] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 196–207, 1999.

[2] A. Avizienis and J. P. J. Kelly. Fault tolerance by design diversity: Concepts and experiments. *IEEE Computer*, 17(8):67–80, 1984.

[3] S. Borkar. Thousand core chips: A technology perspective. In *Conference on Design Automation Conference*, pages 746–749, 2007.

[4] M.R. Choudhury and K. Mohanram. Reliability analysis of logic circuits. *IEEE Transactions on CAD*, 28(3):392–405, 2009.

[5] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[6] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[7] G. Fey, A. Sülflow, S. Frehse, and R. Drechsler. Effective robustness analysis using bounded model checking techniques. *IEEE Transactions on CAD*, 30(8):1239–1252, 2011.

[8] F. Haedicke, S. Frehse, G. Fey, D. Große, and R. Drechsler. metaSMT: Focus on your application not on solver integration. In *International Workshop on Design and Implementation of Formal Tools and Systems*, pages 22–29, 2011.

[9] J. Hayes, I. Polian, and B. Becker. An analysis framework for transient-error tolerance. In *IEEE VLSI Test Symposium*, pages 249–255, 2007.

[10] V. Izosimov, I. Polian, P. Pop, P. Eles, and Z. Peng. Analysis and optimization of fault-tolerant embedded systems with hardened processors. In *Design, Automation and Test in Europe*, pages 682–687, 2009.

[11] D. Kröning. Software verification. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, pages 505–532. IOS Press, 2009.

[12] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–88, 2004.

[13] M. Miskov-Zivanov and D. Marculescu. Circuit reliability analysis using symbolic techniques. *IEEE Transactions on CAD*, 25(12):2638–2649, 2006.

[14] N. Oh, P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1):111–122, 2002.

[15] N. Oh, P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in superscalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

[16] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. In *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 33–42, 2001.

[17] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[18] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray. Low-cost on-line fault detection using control flow assertions. In *IEEE International On-Line Testing Symposium*, pages 137–143, 2003.