Ergebnisvalidierung und nebenläufige Hardwarefehlererkennung mittels systematisch erzeugter Diversität

Heidrun Dücker

Universität Karlsruhe
Institut für Rechnerentwurf und Fehlertoleranz
Postfach 6980, W-7500 Karlsruhe 1
Tel: ++49 721 608 4353
email: duecker@ira.uka.de

Kurzfassung:

Mit steigendem Einsatz von Mikroprozessoren haben sich diese immer mehr auch in sicherheitsrelevanten Gebieten durchgesetzt. Die für solche Einsatzgebiete geforderte hohe Zuverlässigkeit und Sicherheit (im Sinne von engl. safety) kann durch Fehlertoleranzverfahren erreicht werden. Eine Möglichkeit hierzu bietet der Einsatz von Diversität, bei der mehrere Programme erstellt werden, welche die gleiche Spezifikation erfüllen sollen. Dabei ermöglicht ein Vergleich der von einzelnen Programmvarianten berechneten Ergebnisse neben der Entwurfsfehlererkennung auch eine Erkennung permanenter und transienter Hardwarefehler. Entwurfsdiversität alleine garantiert aber keine unterschiedliche Nutzung der Hardware, so daß virtuelle Mehrfachsysteme mit entwurfsdiversitären Programmvarianten nur eine unzureichende Hardwarefehlererkennung aufweisen. Durch den Einsatz von systematisch erzeugter Diversität kann die Hardwarefehlererfassung verbessert werden, da systematisch erzeugte Diversität eine automatische Veränderung des Programmablaufs unter Beibehaltung des implementierten Algorithmus ermöglicht, so daß die Ergebnisse gezielt auf verschiedenen Datenpfaden berechnet werden. Es wurde ein Verfahren zur systematischen Generierung von diversitären Programmvarianten entwickelt, das auf einer diversitären Darstellung der Daten beruht. Zur Realisierung dieses Verfahrens wurde auf der Basis einer komplementären Datenrepräsentation ein Precompiler auf Assemblerebene entwickelt.

Schlüsselwörter:
Diversität, Hardwarefehler, softwareimplementierte Fehlertoleranz, Sicherheit, Testen, Zuverlässigkeit

1. Überblick

Seit vielen Jahren werden Rechensysteme immer mehr in Bereichen eingesetzt, die eine hohe Zuverlässigkeit und Sicherheit erfordern, entweder um Katastrophen oder hohe finanzielle Verluste zu verhindern. Systeme, die eine hohe Zuverlässigkeit und Sicherheit erfordern, benötigen Maßnahmen zur Fehlererkennung, um die Systeme bei Auftreten eines Fehlers entweder in einen sicheren Zustand überführen zu können (fail-safe) oder um andererseits durch ein Fehlertoleranzverfahren trotzdem noch die gewünschte Funktion erbringen zu können.

In sicherheitsrelevanten Systemen sollte sowohl die Hardware als auch die Software zuverlässig funktionieren. Für eine sichere Ausführung eines Programms ist es essentiell, daß Fehler in der Hardware im Falle ihres Auftretens erkannt bzw. toleriert werden. Zwischen dem üblichen Ansatz der auszuführenden Absoluttests, die mitunter nur eine unzureichende Fehlererfassung aufweisen, und dem aufwendigen Ansatz der Hardware-Vervielfachung und Relativtests zwecks Ergebnisvergleich steht der Ansatz der Zeitredundanz: Auf einem einzigen Rechner wird ein Programm mehrfach ausgeführt und auf die Ergebnisse ein Relativtest angewandt. Ein Ansteigen der Verfügbarkeit kann dabei im allgemeinen nicht garantiert werden, die Sicherheit kann aber verbessert werden, wenn ein sicherer Systemzustand bekannt ist [vgl. /Mula 85/].

Damit nicht nur intermittierende, sondern auch permanente Fehler erkennbar werden, empfiehlt sich eine Transformation der Programmvarianten, um diese diversitär zu gestalten, so daß ein permanenter Fehler mit hoher Wahrscheinlichkeit zu einer Ergebnisabweichung führt. Eine solche Transformation kann durch Entwurfsdiversität oder durch systematisch erzeugte Diversität erreicht werden. Unter Entwurfsdiversität versteht man einen unterschiedlichen Entwurf, der durch verschiedene Entwurfsteams durchgeführt

wird und dadurch voneinander verschiedene Implementierungen erreicht. Angestrebt werden hierbei insbesondere verschiedenartige Algorithmen. Systematisch erzeugte Diversität erhält man dagegen durch systematische und automatisch durchführbare Modifikationen des Programm- und/oder Datenflusses unter Beibehaltung desselben Algorithmus.

Wird zur Fehlertoleranz von Softwarefehlern Entwurfsdiversität eingesetzt, kann man durch die Ausführung der diversitären Programmvarianten neben den Entwurfsfehlern auch Hardwarefehler tolerieren /Hinz 89/. Man erhält aber nur eine geringe Hardwarefehlerüberdeckung, die auf eine gleichartige Nutzung der Hardware durch zueinander ähnliche Funktionsblöcke in den einzelnen Programmvarianten zurückzuführen ist. Durch eine zusätzliche systematische Modifikation der entwurfsdiversitären Programmvarianten, durch die die gleichartigen Programmteile diversitär gestaltet werden, kann die Hardwarefehlererkennung erhöht werden. Hierzu wird ein Verfahren vorgestellt, das eine systematische Generierung diversitärer Programmvarianten ermöglicht.

Diese Form der Hardwarefehlererfassung eignet sich nicht für den Produktionstest, da ein großer Aufwand erforderlich ist. Die Fehlererkennung durch Diversität bietet sich dagegen beim Einsatz von Rechensystemen in Bereichen mit sehr hohen Sicherheits- oder Zuverlässigkeitsanforderungen als Online-Testmethode an, wobei der zusätzliche Zeitaufwand erheblich ist, das Verfahren aber auf Standard-Hardware eingesetzt werden kann. Wird Entwurfsdiversität zur Softwarefehlererkennung ohnehin eingesetzt, dann erfordert der Einsatz von systematisch erzeugter Diversität nur noch einen geringen Zusatzaufwand.

2. Theoretische Grundlagen

Hardware- und Softwarefehler können durch folgende Testarten erkannt werden:

Absoluttests: Darunter versteht man Tests, die die Daten bzw. das Rechensystem während des laufenden Betriebs bezüglich vorgegebener Konsistenzbedingungen überprüfen (z. B. Adreßraumüber-

wachung). Die Fehlererfassung von Absoluttests ist von den vorgegebenen Konsistenzbedingungen abhängig. Nur Fehler, durch die Konsistenzbedingungen verletzt werden, können erfaßt werden.

Relativtests: Hierbei werden Ergebnisse mehrfach ausgeführter Funktionen miteinander verglichen. Diese Ergebnisse können je nach eingesetzter Redundanzart parallel oder sequentiell berechnet werden. Durch die Relativtests können beliebige Fehlerarten erkannt werden, aber die Anzahl der zu tolerierenden Fehler ist von der Anzahl der redundanten Funktionsberechnungen abhängig.

Das bei der Erkennung von Software-Entwurfsfehlern am häufigsten eingesetzte Verfahren ist die Verwendung von *Diversität*, bei der verschiedene Programmvarianten entwickelt werden, die dieselbe Spezifikation erfüllen. Danach werden die einzelnen Programmvarianten entweder parallel oder sequentiell ausgeführt:

- Bei der parallelen Ausführung laufen die einzelnen Programmvarianten gleichzeitig auf verschiedener Hardware ab und die Ergebnisse werden anschließend durch einen Relativtest verglichen. Parallele Ausführung erfordert also mehrfache Hardware, benötigt aber kaum Zeitredundanz.

- Bei der sequentiellen Abarbeitung werden die einzelnen Programmvarianten nacheinander ausgeführt. Durch das sequentielle Ausführen der einzelnen Programmvarianten erfordert diese Form der Fehlertoleranz viel Zeitredundanz, aber dafür keine strukturelle Redundanz.

Man unterscheidet Entwurfsdiversität und systematisch erzeugte Diversität:

Unter *Entwurfsdiversität* versteht man einen unterschiedlichen Entwurf, der durch verschiedene Entwurfsteams durchgeführt wird und für den voneinander verschiedene Implementierungen kennzeichnend sind:

- Bei einem *gegensätzlichen Entwurf* treffen die Entwurfsteams Absprachen bezüglich der Realisierung, beispielsweise über die Spezifikationsmethode, die Programmiersprache, den Übersetzer und den Prozessortyp zur Ausführung der Programme. Die

wichtigste Absprache ist aber diejenige über die gegensätzlich zu implementierenden Entwurfsalternativen.

- Bei einem *unabhängigen Entwurf* entwerfen die Teams (ohne jeden Kontakt untereinander) unabhängige Implementierungen.

Systematisch erzeugte Diversität erhält man durch automatisch durchführbare Modifikationen des Programm- und/oder Datenflusses unter Beibehaltung desselben Algorithmus. Es gibt verschiedene Möglichkeiten, durch die Diversität systematisch erzeugt werden kann, von denen nur wenige hier erwähnt werden sollen:

- Eine einfache Möglichkeit der systematisch erzeugten Diversität ist die Vertauschung der Register. Dadurch wirken sich Haftfehler (engl. stuck-at-fault) in den Registern auf unterschiedliche Variablen bei der Berechnung des Ergebnisses aus, so daß damit eine erhöhte Fehlerüberdeckung bei diesen Fehlern erreicht werden kann.

- Eine weitere Möglichkeit für die Realisierung der systematisch erzeugten Diversität bietet die gezielte Untersuchung des Befehlsablaufs auf parallelisierbare Befehlssequenzen. Existieren solche parallelisierbare Sequenzen, so wird die Reihenfolge, mit der sie ausgeführt werden vertauscht. Durch die daraus folgende unterschiedliche Reihenfolge bei der Berechnung der Zwischenvariablen können unter anderem datenabhängige Fehler, d. h. Fehler, die vom Inhalt der umliegenden Speicherbereiche abhängen, leichter erkannt werden.

- Bei häufig vorkommenden Befehlssequenzen werden aus der Kenntnis der Semantik neue, diversitäre Sequenzen entworfen. Diese können durch einen Precompiler unter Berücksichtigung der anwendungsspezifischen Eigenschaften anstelle der ursprünglichen Befehlssequenzen in diversitär zu gestaltende Programmvarianten eingesetzt werden.

- Durch die Verwendung einer von der Standarddarstellung verschiedenen Repräsentation der Daten kann eine Programmvariante systematisch diversitär gestaltet werden. Eine diversitäre Datenrepräsentation kann mit einer Abbildungsfunktion, die auf die Daten angewendet wird, realisiert werden.

Aus der veränderten Datenrepräsentation folgt zusätzlich eine Modifikation der Befehle, d. h. anstelle eines gegebenen Befehls wird eine zugehörige Befehlssequenz ausgeführt, die das gesuchte Ergebnis in der modifizierten Datenrepräsentation berechnet. Wird in einem Programm jeder Befehl bezüglich der Abbildungsfunktion diversitär entwickelt, so erhält man eine Programmvariante, die unter Beibehaltung des implementierten Algorithmus die Ergebnisse mit unterschiedlichen Befehlssequenzen und unterschiedlichem Datenfluß berechnet.

Die ersten drei Varianten der systematisch erzeugten Diversität verwenden nur eine Veränderung des Befehlsablaufs ohne dabei die Daten und deren Darstellung mit in Betracht zu ziehen. Bei der letzten Version werden die Daten des Definitionsbereichs der Befehle auf den Datenbereich modifizierter Befehlssequenzen transformiert, sofern transformierte Daten, modifizierte Befehlssequenz und rücktransformierte Daten die gleichen Ergebnisse wie die ursprüngliche Programmvariante berechnen.

Die angegebenen Methoden ermöglichen alle algorithmisch durchführbare Transformationen, mit denen durch einen Precompiler aus einer gegebenen Programmvariante systematisch ein diversitäres Programm erzeugt werden kann.

Hinweis:
Es können nicht immer scharfe Trennlinien zwischen gegensätzlichem Entwurf und systematisch erzeugter Diversität gezogen werden. Manche Absprachen könnte man als systematisch bezeichnen (das eine Team verwendet Prolog für einen 80486-Prozessor, das andere Pascal für den 68030), da für solche Absprachen Regeln aufgestellt (d. h. im Extremfall sogar Algorithmen angewandt) werden können.
Trotzdem soll aus folgenden Gründen die Unterscheidung zwischen gegensätzlichem Entwurf und systematischer Diversität beibehalten werden:

- Die praktischen Vorgehensweisen unterscheiden sich bei diesen beiden Diversitätsarten stark voneinander.
- Gegensätzlicher Entwurf erkennt/toleriert (vorwiegend Entwurfs-) Fehler in dem entworfenen Subsystem selbst. Dagegen erkennt/toleriert systematisch erzeugte Diversität

(vorwiegend Betriebs-) Fehler in den darunterliegenden Schichten (hauptsächlich Hardwarefehler).

2.1 Das Prinzip der Hardwarefehlererkennung durch diversitäre Software

2.1.1 Ablauf der Fehlererkennung

Hardwarefehler, die sich auf die zu berechnenden Ergebnisse auswirken, können beim Ablauf einer einzelnen Programmvariante (d. h. bei einem Simplexsystem) nur dann erkannt werden, wenn die Fehlerauswirkungen durch die parallel zum Programmablauf durchgeführten Absoluttests erfaßt werden können. Treten aber Ergebnisabweichungen auf, durch die der erlaubte Wertebereich nicht verlassen wird, so können diese Fehler nicht erkannt werden.

In einem *virtuellen Mehrfachsystem* laufen mehrere diversitäre Programmvarianten nacheinander auf einer Hardware ab. Da diversitäre Programmvarianten Befehle und Register nicht in der gleichen Reihenfolge verwenden, können sich die Hardwarefehler bei diesen Programmen an verschiedenen Stellen auswirken, so daß viele der Fehler durch die während der zusätzlichen Programmläufe durchgeführten Absoluttests oder durch den anschließenden Relativtest erkannt werden können (siehe Bild 2.1).

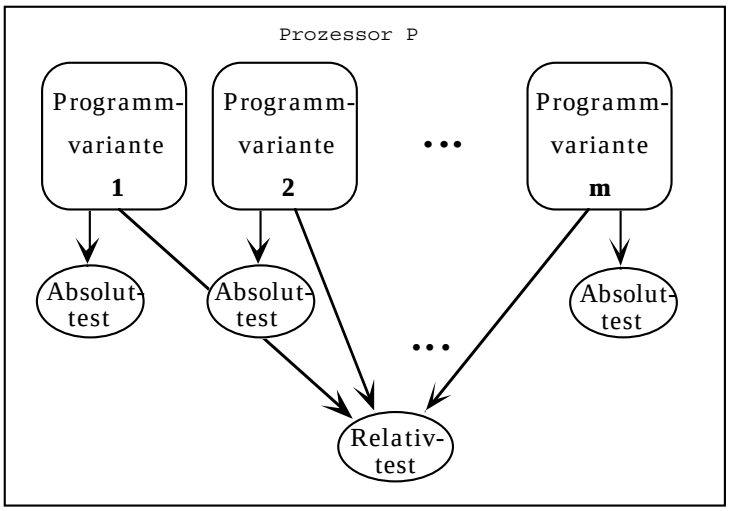

Bild 2.1: Darstellung eines allgemeinen virtuellen Mehrfachsystems

Mit m = 2 erhält man für Systeme mit einer hohen Sicherheitsanforderung ein fehlererkennendes System, mit dem je nach eingesetzter Diversität Hard- und/oder Softwarefehler erkannt werden können.

1) Wird reine systematisch erzeugte Diversität eingesetzt, dann können nur Hardwarefehler erkannt werden, da Entwurfsfehler bei der Generierung der diversitären Variante dupliziert werden. Der Zeitaufwand, der zur Erkennung der Hardwarefehler benötigt wird, ist durch die Verdopplung der Programmdurchführung sehr hoch, so daß andere bekannte Methoden zur Hardwarefehlererkennung sinnvoller sein werden.

2) Bei der Verwendung von reiner Entwurfsdiversität können sowohl Entwurfsfehler in der Software, als auch Hardwarefehler erkannt werden, wobei die Hardwarefehlerüberdeckung allerdings nur gering ist /EHNi 90/.

3) Durch eine Kombination aus systematischer Diversität und Entwurfsdiversität bleibt die Softwarefehlererkennung erhalten und die Hardwarefehlerüberdeckung wird erhöht. In Bild 2.2 ist ein virtuelles Duplexsystem dargestellt, bei dem eine von zwei entwurfsdiversitären Varianten noch systematisch verändert wurde, um die Hardwarefehlererfassung zu erhöhen.

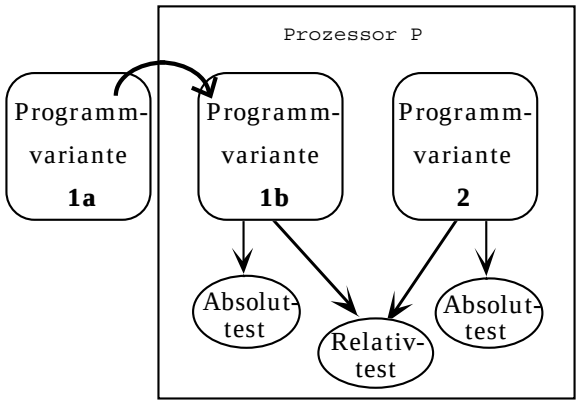

Bild 2.2:Darstellung eines virtuellen Duplexsystems aus systematisch erzeugter und entwurfsdiversitären Programmvarianten

Da ein fehlererkennendes System nicht zur Erhöhung der Zuverlässigkeit eingesetzt werden kann, muß man für Systeme mit einer hohen Zuverlässigkeitsanforderung mit diesem Prinzip ein fehlertolerierendes n-aus-m System (d. h. m > 2) entwickeln. Die größte Fehlerüberdeckung erhält man wie bei den fehlererkennenden

Systemen bei einer Kombination aus systematischer Diversität und Entwurfsdiversität. Dabei werden die verschiedenen entwurfsdiversitären Varianten V_i durch unterschiedliche Realisationen von systematisch erzeugter Diversität modifiziert (siehe Bild 2.3).

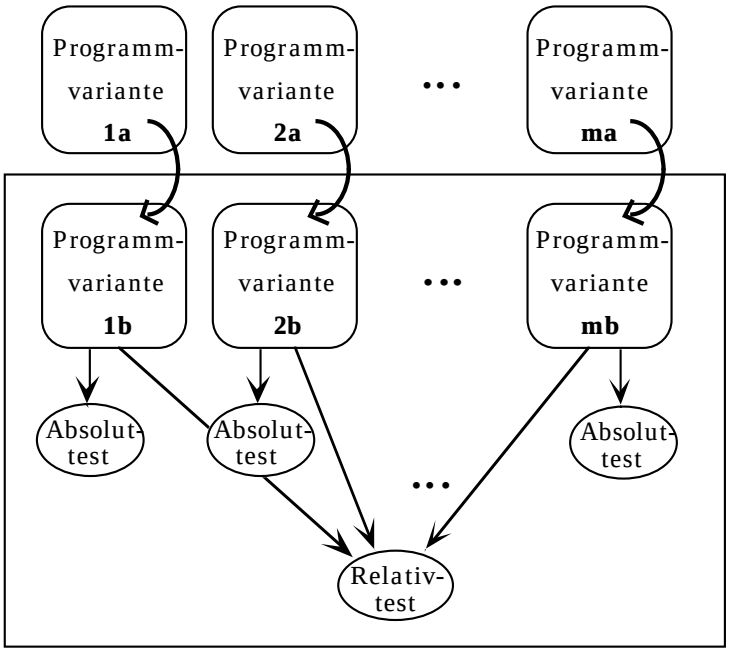

Bild 2.3: Darstellung eines virtuellen Mehrfachsystems aus systematisch und entwurfsdiversitären Programmvarianten

2.1.2 Fehlermodell

Im verwendeten Fehlermodell werden für die Prozessor-Hardware permanente Einzel-Haftfehler betrachtet. Dabei werden aber nur die prozessorabhängigen Fehler untersucht, da Speicherfehler leicht durch eine geeignete Codierung erkannt werden können. Die prozessorabhängigen Fehler können dabei, bezogen auf die Mikroprozessoren der 68000-Familie von Motorola, in drei Fehlertypen unterteilt werden:

Busfehler: bei dem eine einzelne Leitung des Busses permanent auf einem Wert 0 oder 1 liegt.

Registerfehler: bei dem ein einzelnes Bit eines Registers ständig auf 1 oder 0 liegt.

Instruktionsfehler: bei dem eine falsche Ausführung einer einzelnen Instruktion des Befehlssatzes modelliert

wird. D. h. anstelle der Instruktion I wird immer eine andere Instruktion J ausgeführt.
= durch Steuerwerksfehler bedingte Ausführung eines vom Programm abweichenden Befehls.

Die Fehlerüberdeckung bezüglich der oben klassifizierten Fehler wird mit einem softwareimplementierten Fehlerinjektor /Hinz 89/, der für den 68000-Prozessor von Motorola an der Universität Karlsruhe verfügbar ist, erfaßt. Dieser Fehlerinjektor erzeugt die modellierten Fehlereffekte durch die Programmausführung im Einzelschrittmodus für jeden Programmbefehl und überprüft nach vollständiger Programmausführung der diversitären Varianten, ob die Fehler erkannt werden können.

Die Ergebnisse können bei der Auswertung folgenden Ergebnisklassen zugeordnet werden:

1) korrekt

2) fehlerhaft; erkannt durch Absoluttest

3) fehlerhaft; erkannt durch Ergebnisvergleich

4) gefährlich fehlerhaft; nicht erkannt, d. h. die Anzahl der identisch falschen Ergebnisse überwiegt.

Bei einem Simplexsystem existiert die Ergebnisklasse 3 nicht, da nur ein Ergebnis vorliegt. Die Ergebnisklasse 4 tritt immer dann ein, wenn ein falsches Endergebnis berechnet wird.

Der Einsatz von Entwurfsdiversität ermöglicht neben der Softwarefehlererkennung auch eine geringe Hardwarefehlererkennung, die zu einer Reduzierung des Auftretens des gefährlich fehlerhaften Zustandes führt. Durch den zusätzlichen Einsatz von systematischer Diversität werden weitere Hardwarefehler erkennbar, die ursprünglich zu einem gefährlich fehlerhaften Zustand führten.

2.2 Verfahren zur Generierung von systematischer Diversität

Basiert die Diversität auf der Modifikation der Datenrepräsentation, müssen für die auszuführenden Befehle unter Berücksichtigung der verwendeten Datenrepräsentation sowie des verwendeten Prozessors neue Befehlssequenzen entworfen, durch die zu den Originalbefehlen gehörigen Ergebnisse in der neuen Datenrepräsentation berechnet werden. Bezüglich einer eindeutigen Datenrepräsentation R stehen die einzelnen Befehle der diversitären Programmvarianten zueinander in folgender Beziehung:

Definition 1:

Sei ein Prozessor P mit einem Datenbereich D von darstellbaren Daten und einem Befehlssatz $F = \{f_i: DxD \rightarrow W \subseteq D\}$ gegeben.

Es existiere weiterhin zu einer Datenrepräsentation R eine bijektive, algorithmisch berechenbare Abbildung $r: D \rightarrow D$ (sowie deren Inverse r^{-1}), so daß zu jedem Befehl f_i eine Befehlssequenz s_i von Befehlen aus F mit folgenden Eigenschaften realisierbar ist:

$$s_i : r(D) \times r(D) \rightarrow r(W)$$

$$\text{mit} \quad s_i(r(a) , r(b)) = r(f_i(a,b)) \quad \text{und} \quad a,b \in D.$$

Die zum Befehl f_i diversitäre Befehlssequenz bezüglich der Datenrepräsentation R ist dann durch $r^{-1} \circ s_i \circ r$ gegeben.

Eine andere Methode miteinander vergleichbare diversitäre Sequenzen zu erhalten, ist folgende Modifikation beider Varianten: $s \circ r$ und $r \circ f$.

Eine diversitären Befehlssequenz zu einem Befehl f, der aus den Operanden a und b das Ergebnis f(a,b) berechnet, benötigt die im folgenden beschriebenen drei Schritte, um aus den gegebenen Operanden a und b mittels diversitärer Datenrepräsentation das gesuchte Ergebnis f(a,b) zu berechnen (siehe Bild 2.4):

1. Transformation der Operanden durch die Abbildungsfunktion r

2. Ausführung einer speziellen Befehlssequenz s, die die transformierten Operanden in die Transformation des gewünschten Ergebnisses überführt.

3. Rücktransformation des Ergebnisses durch die Umkehrfunktion r^{-1}

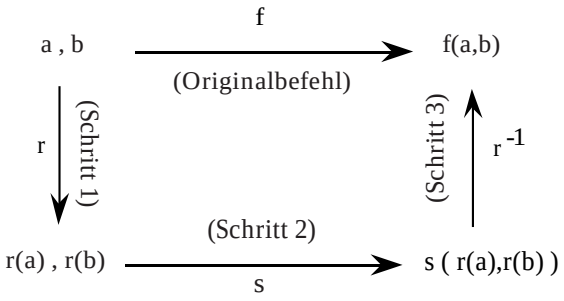

Bild 2.4: Ablauf einer diversitären Befehlssequenz

Bei einer Folge von Befehlen wird jeder einzelne Befehl entsprechend der gegebenen Abbildungsfunktion r verändert. Dabei können Schritt 3 des Befehls f_i und Schritt 1 des Befehls f_{i+1} zwischen den einzelnen Befehlen der Sequenz entfallen (siehe Bild 2.5), da die Zwischenvariablen nur von der modifizierten Datenrepräsentation in die Standardrepräsentation und wieder in die modifizierte Darstellung transformiert werden.

Dies gilt, da die Rücktransformation eines Wertes r(x) durch $r^{-1}(r(x))$ gegeben ist und für die erneute Transformation dieses Wertes in die modifizierte Datenrepräsentation gilt:

$$r(r^{-1}(r(x))) = r(x).$$

Bild 2.5 gibt den Ablauf eines Programms und der zugehörigen systematisch generierten diversitären Programmvariante wieder. Wenn das Ursprungsprogramm aus den Eingangsdaten E über die Zwischenergebnisse Z_i der Befehle f_i die Ausgabedaten A berechnet, dann kann die diversitäre Programmvariante die diversitär dargestellten Ausgabedaten A mit den modifizierten Befehlssequenzen s_i über die Zwischenergebnisse $r(Z_i)$ berechnen. Dabei können die Datentransformationen zwischen den Befehlssequenzen s_i (Schritt 3) und s_{i+1} (Schritt 1) entfallen.

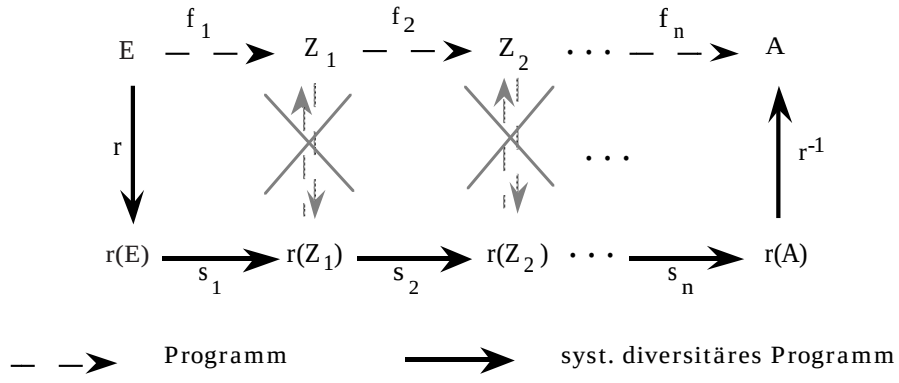

Bild 2.5: Ablauf eines systematisch diversitären Programms

3 Realisierung des Verfahrens

3.1 Wahl der Abbildungsfunktion r

Prinzipiell kann jede bijektive Abbildung r: D → D, für die es zu jedem Befehl eines Prozessors eine diversitäre Befehlssequenz gibt, zur systematischen Entwicklung von Diversität eingesetzt werden. Für die Effizienz dieses Verfahrens muß die Datenabbildungsfunktion r leicht zu generieren sein und die Befehlssequenzen s_i müssen mit dem Befehlssatz des Prozessors effizient realisierbar sein. Außerdem sollte durch den Einsatz der durch die Abbildungsfunktion r gegebenen systematisch erzeugten Diversität die Hardwarefehlererfassung möglichst hoch sein.

Die bekannte Erfassung tatsächlich auftretender Fehler mit dem Haftfehlermodell sollte durch eine geeignete Wahl der Abbildung r aufgegriffen werden. Ein bekanntes Verfahren, das bei Speichern eingesetzt wird, komplementiert die gespeicherten Daten zur Unterscheidung transienter und permanenter Fehler. Das Komplementieren der Daten hat den Vorteil, daß die Anzahl der darstellbaren Daten durch diese Abbildung nicht verändert wird, wie es z. B. durch die von Hahn, Gössel und Vermeiren in [HaGö 91] verwendete Verschiebung der Daten um eine Bitstelle nach links der Fall ist. Desweiteren ist die Komplementierung der Daten für r und r^{-1} leicht durchführbar.

Es gibt zwei bekannte Verfahren, mit denen Zahlen komplementär dargestellt werden können: das Einer- und das Zweierkomplement. Eine einfache Realisierung der Diversität durch die Verwendung des Komplements erfordert für jeden Befehl f effiziente Realisierung der Befehlssequenz s. Um nun eine Entscheidung zwischen den beiden Komplementarten treffen zu können, müssen jene Befehlsklassen näher untersucht werden, die eine Datenmanipulation vornehmen. Dies sind im wesentlichen die arithmetischen und die logischen Befehle.

Logische Befehle können mit den de Morgan´schen Regeln bezüglich der Komplementbildung umgeformt werden. Für die logische Verknüpfung ODER gilt nach de Morgan:

$$a \vee b = \overline{\overline{a \vee b}} = \overline{\overline{a} \wedge \overline{b}}$$

Überträgt man diese Form auf den log. Befehl OR a,b, dann erhält man den zu OR diversitären Befehl AND, der aus den invertierten Operanden $\overline{a}$ und $\overline{b}$ das zu c = OR a,b invertierte Ergebnis berechnet:

$$\overline{c} = \text{OR } \overline{a}, \overline{b}$$

Analog lassen sich die de Morgan´schen Regeln auf alle logischen Operationen eines Prozessors übertragen.

Bei den arithmetischen Befehlen bewirkt die Invertierung der einzelnen Bits eines Datenwortes x (= Bildung des Einerkomplements) folgende Wertänderung:

$$\overline{x} = -1 - x.$$

Betrachtet man den arithmetischen Befehl "Division mit Vorzeichen" näher, dann muß für die zugehörige diversitäre Befehlssequenz

$$\overline{\left(\frac{x}{y}\right)} = (-1) - \frac{x}{y} \stackrel{!}{=} \frac{(-1-x)}{(-1-y)}_{\text{diversitär}} = \frac{\overline{x}}{\overline{y}}_{\text{diversitär}}$$

gelten. Die modifizierte Befehlssequenz $s = \frac{(-1-x)}{(-1-y)}_{\text{diversitär}}$ läßt sich folgendermaßen berechnen:

$$\overline{\left(\frac{x}{y}\right)} = -1 - \frac{x}{y} = -1 - \frac{-1-\overline{x}}{-1-\overline{y}} = -1 + \frac{1}{-1-\overline{y}} + \frac{\overline{x}}{-1-\overline{y}} =$$

$$= -1 + \frac{1}{-1-(-1-y)} + \frac{1}{\frac{-1}{-1-x} - \frac{-1-y}{-1-x}}$$

Es kann also $\frac{\overline{x}}{\overline{y}}$ diversitär durch

$$-1 + \frac{1}{-1-\overline{y}} + \frac{1}{\frac{-1}{\overline{x}} - \frac{\overline{y}}{\overline{x}}} = -1 + \frac{1}{-1-(-1-y)} + \frac{1}{\frac{-1}{-1-x} - \frac{-1-y}{-1-x}}$$

ersetzt werden, so daß die geforderte Gleichung erfüllt ist. Diese Berechnung ist sehr aufwendig und ist sowohl für x = –1 als auch für y = 0 nicht definiert, während die ursprüngliche Division nur für y = 0 nicht definiert ist.

Daraus kann man ersehen, daß sich die Invertierung der einzelnen Bitstellen eines Datenwortes nicht immer als geeignete Abbildung für die diversitäre Repräsentation der Daten erweist. Für die arithmetischen Operationen ist die Zweierkomplementdarstellung, bei der außer der Invertierung der einzelnen Bitstellen eine Addition von 1 zusätzlich durchgeführt wird, eine effizientere Abbildungsfunktion. Andererseits ist die Darstellung der Daten im Zweierkomplement nicht für alle logischen Befehle geeignet, so daß in der Abbildungsfunktion r das Einer- und das Zweierkomplement realisiert werden muß.

3.2 Optimierungsverfahren

3.2.1 Aktualisierung des Statusregisters

Nach der Wahl der Datenabbildung müssen für alle Befehle des Prozessors die Befehlssequenzen s entwickelt werden. Dabei muß bei der Untersuchung der Befehle beachtet werden, daß sich die Befehle nicht nur auf die explizit im Befehl angegebenen Operanden auswirken, sondern auch implizit auf das Benutzerbyte des *Statusregisters* (siehe Bild 3.1).

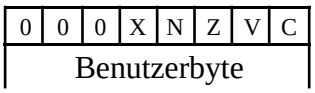

Bild 3.1 Aufbau des Benutzterbytes des Statusregisters des Prozessors M68000

Das Benutzerbyte enthält die Sprungbedingungen (Flags) und wird daher auch als Bedingungsregister (Condition-Code-Register – CCR) bezeichnet. In diesem Byte sind Bit 0 - 4 mit Flags belegt, Bit 5 - 7 sind frei und können zwar beschrieben werden, beim Lesen erhält man jedoch bei diesen Bits immer eine Null. Die Bedingungsbits 0 - 4 werden durch den Operanden bzw. das Ergebnis einer Operation gesetzt oder zurückgesetzt und dienen z. B. als Sprungbedingungen bei Verzweigungen. Welche Bits verändert werden ist dabei vom ausgeführten Befehl abhängig und nicht jeder Befehl modifiziert alle Bedingungsbits.

Bei der Transformation eines Assembler-Programms in ein systematisch diversitäres Programm werden die einzelnen Befehle durch neue Befehlssequenzen ersetzt. Dabei sollen die Bedingungsbits des Statusregisters im Vergleich zu den im Originalprogramm gesetzten Bits invertiert gesetzt werden. Einige der diversitären Befehlssequenzen erfüllen diese Forderung automatisch. Aber es gibt auch Befehlssequenzen, die die Bedingungsbits so verändern, daß die gesetzten Bits im Originalprogramm und im modifizierten Programm in keiner Beziehung zueinander stehen. Für einen nachfolgenden Verzweigungsbefehl, der eines oder mehrere Bedingungsbits abfragt, muß aber eine Beziehung zwischen den Bedingungsbits im Originalprogramm und dem diversitären Programm bestehen, damit die Verzweigung korrekt ausgeführt werden kann. Deshalb muß das Statusregister nach Befehlssequenzen, durch die die Bedingungsbits nicht korrekt gesetzt werden, aktualisiert werden.

Zu jedem Befehl muß also eine Befehlssequenz entwickelt werden, die einerseits die komplementären Daten generiert, und die andererseits die einzelnen Bits des Statusregisters korrekt setzt. Eine solche Generierung der korrekten Flags erfordert zur Laufzeit einen hohen Zeitaufwand, der aber prinzipiell nur dann benötigt wird, wenn

anschließend auf ein nicht korrekt gesetztes Flags zugegriffen wird. Deshalb muß ein Kriterium entwickelt werden, anhand dessen entschieden wird, bei welchen Programmsequenzen die Flags aktualisiert werden müssen und bei welchen nicht.

Würde jeder ausgeführte Befehl alle Bedingungsbits verändern, dann müßten die Werte der Bedingungsbits nur dann aktualisiert werden, wenn mit dem nächsten auszuführenden Befehl auf die Bedingungsbits zugegriffen wird. Es hat sich aber gezeigt, daß viele Befehle nur einen Teil der Bedingungsbit verändern. Zu jedem Befehl muß also solange eine Aktualisierung des Statusregisters in Betracht gezogen werden, wie noch nicht alle Bedingungsbits überschrieben wurden. Gleichzeitig müssen auch für die nachfolgenden Befehle die Bedingungsbits korrekt bearbeitet werden. Die Speicherung der durch nachfolgende Befehle überschriebenen Bedingungsbits wird folglich für mehrere Befehle parallel durchgeführt.

Die Verwendung eines Kriteriums, bei dem die Statusbits nur dann aktualisiert werden, wenn auf die Bits anschließend zugegriffen wird, reduziert den benötigten Zusatzaufwand drastisch, da Befehle, die auf die Bedingungsbits zugreifen, die Minderheit der Befehle eines Programms ausmachen.

3.2.2 Umschaltung zwischen der Verwendung des Einer- bzw. des Zweierkomplements

Wie schon in Kapitel 3.1 erläutert wurde, erfordern die verschiedenen Befehlstypen für eine effiziente Realisierung der beschriebenen systematisch erzeugten Diversität den Einsatz des Einer- und des Zweierkomplements. Je nach Befehlsart wird dann zwischen den beiden Komplementarten umgeschaltet werden. Dabei ist von Vorteil, daß eine Umwandlung zwischen dem Einer- und dem Zweierkomplement leicht durch eine Addition bzw. Subtraktion von "1" durchgeführt werden kann.

Da eine häufige Umwandlung der Datenformate insgesamt zu einem erheblichen Aufwand führt, wird noch untersucht, ob die verschiedenen Befehle auch mit der jeweils anderen Komplementart

realisiert werden können. Für die meisten Befehle gibt es für beide Komplementarten modifizierte Befehlssequenzen s, wobei je nach Befehlstyp eine Modifikation effizienter ist als die andere. Für einige Befehle ist zudem die Befehlssequenz s in der weniger geeigneten Komplementdarstellung effizenter als die Umschaltung zwischen den Datendarstellungen und die Ausführung der schnelleren Befehlssequenz.

Eine einzelne Addition von ganzzahligen Werten zwischen vielen logischen Operationen läßt sich z. B. effizienter im Einerkomplement berechnen als durch die Umschaltung ins Zweierkomplement. Dies erkennt man daran, daß für die Addition im Einerkomplement nur eine zusätzliche Ausgleichsaddition benötigt wird, während für die Umschaltung zwei Additionen und eine Subtraktion benötigt wird (vgl. Tabelle 3.1).

f	s für das Einerkomplement	Transformation	s für das Zweierkomplement

add a,b	add $\overline{a}$, $\overline{b}$
	add #1, $\overline{b}$
		add #1, $\overline{a}$
		add #1, $\overline{b}$
		
		sub #1, $\overline{b}$
			add –a,–b

Tabelle 3.1: Aufwandsvergleich bei einer Umsetzung des Additionsbefehls

Sind einzelne solcher Befehle in eine Befehlssequenz eingebettet, die mit der für diese Befehle weniger geeigneten Komplementdarstellung modifiziert werden (siehe Bild 3.2b), dann ist es bei einem solchen Befehlsablauf effizienter, wenn diese einzelnen Befehle mit der Komplementart der Sequenz realisiert werden, da einzelne Befehle von Transformationen umschlossen sind.

Für große, hintereinander geschaltete Blöcke aus gleichartigen Befehlen, d. h. aus Befehlen, die sich mit der gleichen Komplementdarstellung umformen lassen (siehe Bild 3.2a), ist bei einem Wechsel der Befehlssequenz mit gleichartiger Datenabbildung die Transformation der Daten in die andere Komplementart eine effiziente Lösung. Da die Transformation nur beim Sequenzwechsel durchge-

führt wird, nimmt der Komplementwechsel nur einen kleinen Anteil an der Anzahl der durchgeführten Befehle ein.

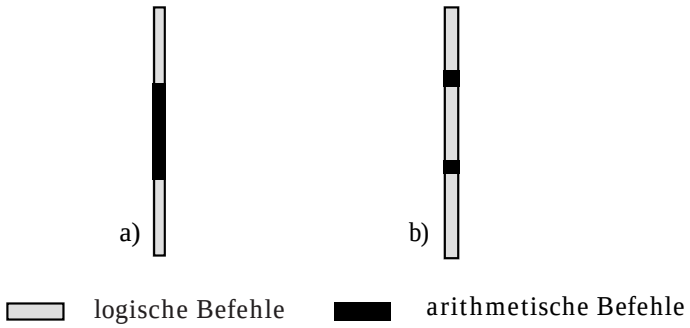

Bild 3.2: Darstellung von Befehlsabläufen:
a) große Befehlssequenzen mit jeweils gleichartiger Datenabbildung
b) Befehlssequenz mit gleichartiger Datenabbildung, die durch einzelne Befehle andersartiger Datenabbildung unterbrochen wird

Abhängig von den Befehlen eines zu modifizierenden Programms kann entschieden werden, ob die Verwendung der aktuellen Komplementdarstellung zur Transformation eines Befehls eine effiziente Lösung ist oder ob eine Umschaltung zur anderen Komplementarten sinnvoller ist.

Allgemein kann man durch die beschriebene Verwendung des Einer- und des Zweierkomplements mehrere verschiedene, diversitäre Programmvarianten generieren. Das Ziel der Umschaltung zwischen den einzelnen Komplementarten ist es aber, eine Programmvariante zu erhalten, deren Zeitaufwand zur Laufzeit möglichst gering ist.

Die verschiedenen diversitären Programmvarianten zu einem Programm P kann man aus dem zugehörigen Transformationsgraphen TG_P (siehe Bild 3.3) gewinnen.

Die Knoten $K_{0,i}$, $i \in \{1,...,n\}$, des Transformationsgraphen entsprechen dabei den Zuständen des Originalprogramms vor der Ausführung des Befehls f_i.

Die Knoten $K_{d,i}$, $d \in \{1,2\}$ und $i \in \{1,...,n\}$, des Transformationsgraphen entsprechen dabei den Zuständen des diversitären Programms, bei denen der letzte Befehl f_{i-1} mit dem Einerkomple-

ment (d=1) bzw. mit dem Zweierkomplement (d=2) transformiert wurde.

Die Kanten $s_{d,i}$, $d \in \{1,2\}$ und $i \in \{1,...,n\}$, des Transformationsgraphen entsprechen den modifizierten Befehlssequenzen zu den Befehlen f_i mit der Repräsentation der Operanden im Einerkomplement (d=1) bzw. im Zweierkomplement (d=2).

Die Kanten $t_{d,i}$, $d \in \{1,2\}$ und $i \in \{1,...,n\}$, des Transformationsgraphen entsprechen den Transformationen der Operanden des Befehls f_i von der Einerkomplement- in die Zweierkomplementdarstellung (d=1) und umgekehrt (d=2).

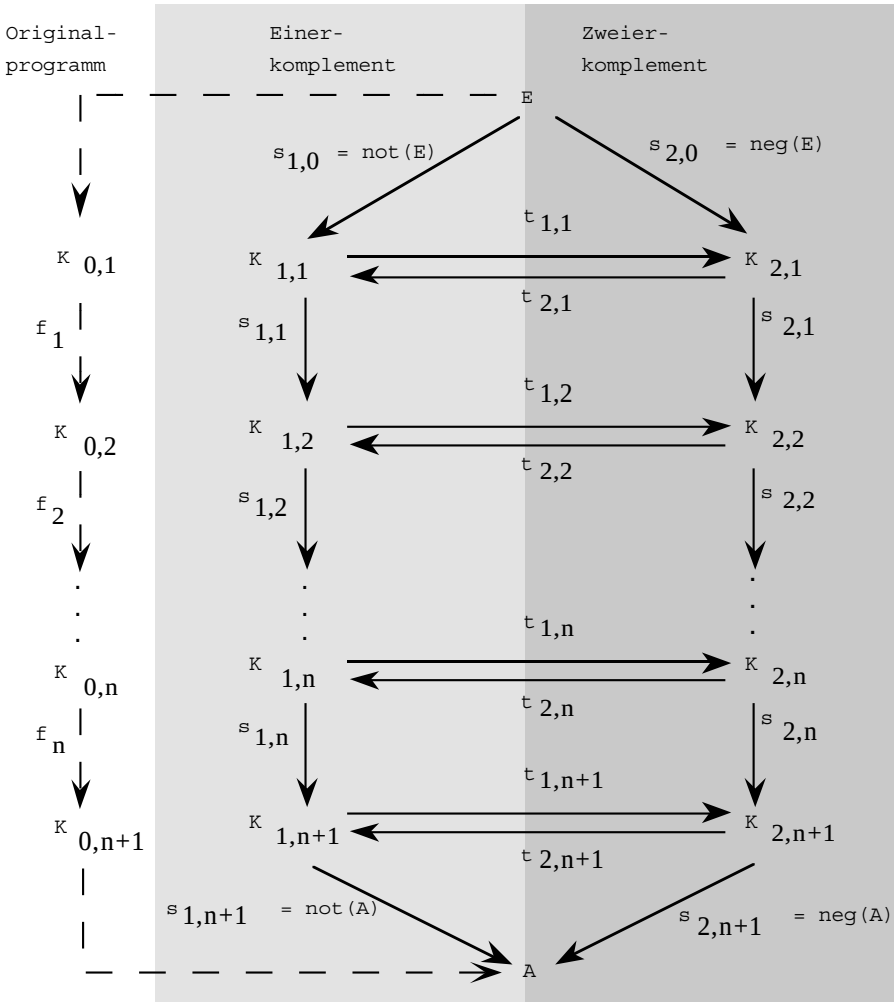

Bild 3.3: Transformationsgraph eines Programms mit Eingabedaten E und Ausgabedaten A

Das Assemblerprogramm einer diversitären Programmvariante entspricht einem Weg durch den Graphen von E nach A, bei dem jeder Knoten des Graphen maximal einmal durchlaufen wird. Ein solcher Weg durch den Transformationsgraphen entspricht einem Subgraphen von TG_P und wird im folgenden $TG_P´$ bezeichnet. Dieser Subgraph $TG_P´$ (siehe Bild 3.4) enthält auf jeder Stufe i genau eine Befehlssequenz $s_{d,i}$ und höchtens ein Transformationssequenz $t_{d,i}$.

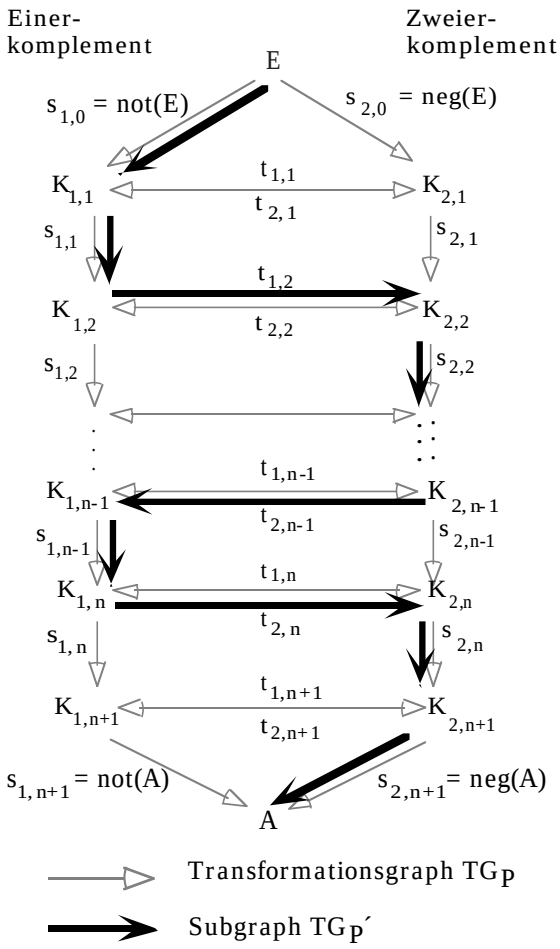

Bild 3.4: Subgraph $TG_P´$ zu einem diversitären Assemblerprogramm

Für ein diversitäres Programm zu einem Ursprungsprogramm P mit den Befehlen f_i, $i \in \{1,...,n\}$, läßt sich folgende Aufwandsformel angeben:

Definition 2:

Die Transformationsfunktion trans(i) gebe an, mit welcher Datenrepräsentation (Einer- oder Zweierkomplement) die einzelnen Befehle f_i transformiert werden und sie sei folgendermaßen definiert:

$$\text{trans}(i) = \begin{cases} 1 & \text{falls } s_{1,i} \text{ Kante von } TG_P' \\ & \text{für } i = 0, \ldots, n+1 \\ 0 & \text{falls } s_{2,i} \text{ Kante von } TG_P' \end{cases}$$

$\text{trans}(-1) = \text{trans}(0)$.

Definition 3:

A(x) bezeichne die Anzahl der für die Ausführung einer Befehlssequenz x benötigten Taktzyklen.

$A(TG_P')^i$ bezeichne den Zeitaufwand der zu den Befehlen $f_1,\ldots,f_i$ modifizierten Befehlssequenzen des zu einem Subgraphen TG_P' gehörigen diversitären Programms. $A(TG_P')^0$ bezeichne den Zeitaufwand, der für die Transformation der Eingabedaten E benötigt wird.

Satz:

Es sei TG_P der Transformationsgraph zu einem Programm P aus n Befehlen. Für den benötigten Zeitaufwand $A(TG_P')$ eines zu einem Subgraphen TG_P' gehörigen diversitären Programms gilt:

$$\begin{aligned} A(TG_P') &= A(TG_P')^{n+1} \\ &= A(TG_P')^n + \\ &\quad + (1-\text{trans}(n+1))\cdot[\text{trans}(n)\cdot A(t_{1,n+1}) + A(s_{2,n+1})] \\ &\quad + \text{trans}(n+1)\cdot[(1-\text{trans}(n))\cdot A(t_{2,n+1}) + A(s_{1,n+1})] \end{aligned}$$

mit $A(TG_P')^0 = \text{trans}(0)\cdot A(s_{2,0}) + \text{trans}(0)\cdot A(s_{1,0})$.

Beweis:

Der Beweis gliedert sich in zwei Teile:

1. Für den Aufwand eines zu P gehörigen diversitären Programms ohne Rücktransformation der Ergebnisse gilt:

$$A(TG_P')^n = A(TG_P')^{n-1} + \\ + (1\text{-trans}(n))\cdot[\text{trans}(n-1)\cdot A(t_{1,n}) + A(s_{2,n})] \\ + \text{trans}(n)\cdot[(1\text{-trans}(n-1))\cdot A(t_{2,n}) + A(s_{1,n})]$$

2. Der Aufwand für die Rücktransformation der Ergebnisse ist gegeben durch:

$$(1\text{-trans}(n+1))\cdot[\text{trans}(n)\cdot A(t_{1,n+1}) + A(s_{2,n+1})] \\ + \text{trans}(n+1)\cdot[(1\text{-trans}(n))\cdot A(t_{2,n+1}) + A(s_{1,n+1})]$$

Es gilt also: $A(TG_P') = A(TG_P')^{n+1} = A(TG_P')^n + A(\text{Rücktrans.})$

Zeige Punkt 1 durch Induktion über die Anzahl der Befehle des Programms P:

Induktionsanfang:

Der Transformationsgraph zu einem minimalen Programm, (n=1), enthält genau 3 Ebenen (Transformation der Eingabewerte, modifizierte Befehlssequenz zum Befehl f_1 und Rücktransformation der Daten).

<u>1. Fall</u>: $s_{1,0}$ und $s_{1,1}$ sind Kanten von $TG_P' \Rightarrow \text{trans}(0) = \text{trans}(1) = 1$

$A(TG_P')^0 = 0\cdot A(s_{2,0}) + 1\cdot A(s_{1,0}) = A(s_{1,0})$

$A'(TG_P')^1 = A(TG_P')^0 + 0\cdot[\ldots] + 1\cdot[0\cdot A(t_{2,1}) + A(s_{1,1})]$

$= A(s_{1,0}) + A(s_{1,1})$

<u>2. Fall</u>: $s_{1,0}$ und $s_{2,1}$ sind Kanten von $TG_P' \Rightarrow \text{trans}(0) = 1, \text{trans}(1) = 0$

$A(TG_P')^0 = 0\cdot A(s_{2,0}) + 1\cdot A(s_{1,0}) = A(s_{1,0})$

$A'(TG_P')^1 = A(TG_P')^0 + 1\cdot[1\cdot A(t_{1,1}) + A(s_{2,1})] + 0\cdot[\ldots]$

$= A(s_{1,0}) + A(t_{1,1}) + A(s_{2,1})$

<u>3. Fall</u>: $s_{2,0}$ und $s_{2,1}$ sind Kanten von $TG_P' \Rightarrow \text{trans}(0) = \text{trans}(1) = 0$

$A(TG_P')^0 = 1\cdot A(s_{2,0}) + 0\cdot A(s_{1,0}) = A(s_{2,0})$

$A'(TG_P')^1 = A(TG_P')^0 + 1\cdot[0\cdot A(t_{1,1}) + A(s_{2,1})] + 0\cdot[\ldots]$

$= A(s_{2,0}) + A(s_{2,1})$

<u>4. Fall</u>: $s_{2,0}$ und $s_{1,1}$ sind Kanten von $TG_P' \Rightarrow \text{trans}(0) = 0, \text{trans}(1) = 1$

$A(TG_P')^0 = 1\cdot A(s_{2,0}) + 0\cdot A(s_{1,0}) = A(s_{2,0})$

$A'(TG_P')^1 = A(TG_P')^0 + 0\cdot[\ldots] + 1\cdot[1\cdot A(t_{2,1}) + A(s_{1,1})]$

$= A(s_{2,0}) + A(t_{2,1}) + A(s_{1,1})$

$\Rightarrow$ Die Gleichung für $A(TG_P')^n$ ist für n = 1 erfüllt.

Induktionsannahme:

Sei die Gleichung für ein beliebiges $n \geq 1$ erfüllt, d. h. es gilt:

$$A(TG_P')^n = A(TG_P')^{n-1} + (1-trans(n))\cdot[trans(n-1)\cdot A(t_{1,n}) + A(s_{2,n})] + trans(n)\cdot[(1-trans(n-1))\cdot A(t_{2,n}) + A(s_{1,n})]$$

Induktionsschritt:

Für $A(TG_P')^{n+1}$ gilt dann:

1. Fall: $s_{1,n}$ und $s_{1,n+1}$ sind Kanten von TG_P'
$\Rightarrow trans(n) = trans(n+1) = 1$
Da die Befehle f_n und f_{n+1} mit dem Einerkomplement transformiert werden, ergibt sich der Zeitaufwand $A(TG_P')^{n+1}$ aus dem bis zum Befehl f_n benötigten Aufwand $A(TG_P')^n$ plus der zusätzlichen Zeit für die Befehlssequenz $s_{1,n+1}$.
$\Rightarrow A(TG_P')^{n+1} = A(TG_P')^n + A(s_{1,n+1})$
Nach der Gleichung für $A(TG_P')$ gilt:
$A(TG_P')^{n+1} = A(TG_P')^n + 0 \cdot[...] + 1 \cdot[0 \cdot A(t_{2,n}) + A(s_{1,n+1})]$
$= A(TG_P')^n + A(s_{1,n+1})$

2. Fall: $s_{1,n}$ und $s_{2,n+1}$ sind Kanten von TG_P'
$\Rightarrow trans(n) = 1,\ trans(n+1) = 0$
Da die Befehle f_n und f_{n+1} mit unterschiedlichen Komplementarten transformiert werden, ergibt sich der Zeitaufwand $A(TG_P')^{n+1}$ aus dem bis zum Befehl f_n benötigten Aufwand $A(TG_P')^n$ plus der zusätzlichen Zeit für die Transformation der Operanden in das Zweierkomplement sowie der Zeit für die Befehlssequenz $s_{2,n+1}$.
$\Rightarrow A(TG_P')^{n+1} = A(TG_P')^n + A(t_{1,n}) + A(s_{2,n+1})$
Nach der Gleichung für $A(TG_P')$ gilt:
$A(TG_P')^{n+1} = A(TG_P')^n + 1 \cdot[1 \cdot A(t_{1,n}) + A(s_{2,n+1})] + 0 \cdot[...]$
$= A(TG_P')^n + A(t_{1,n}) + A(s_{2,n+1})$

3. Fall: $s_{2,n}$ und $s_{2,n+1}$ sind Kanten von TG_P'
$\Rightarrow trans(n) = trans(n+1) = 0$

Der Zeitaufwand $A(TG_P')^{n+1}$ ergibt sich aus dem bis zum Befehl f_n benötigten Aufwand $A(TG_P')^n$ plus der zusätzlichen Zeit für die Befehlssequenz $s_{2,n+1}$.
$\Rightarrow A(TG_P')^{n+1} = A(TG_P')^n + A(s_{2,n+1})$
Nach der Gleichung für $A(TG_P')$ gilt:

$$A(TG_P')^{n+1} = A(TG_P')^n + 1 \cdot [0 \cdot A(t_{1,n}) + A(s_{2,n+1})] + 0 \cdot [...]$$
$$= A(TG_P')^n + A(s_{2,n+1})$$

4. Fall: $s_{2,n}$ und $s_{1,n+1}$ sind Kanten von TG_P'
$\Rightarrow trans(n) = 0,\ trans(n+1) = 1$
Der Zeitaufwand $A(TG_P')^{n+1}$ ergibt sich aus dem bis zum Befehl f_n benötigten Aufwand $A(TG_P')^n$ plus der zusätzlichen Zeit für die Transformation der Operanden in das Zweierkomplement sowie der Zeit für die Befehlssequenz $s_{1,n+1}$.
$\Rightarrow A(TG_P')^{n+1} = A(TG_P')^n + A(t_{2,n}) + A(s_{1,n+1})$
Nach der Gleichung für $A(TG_P')$ gilt:

$$A(TG_P')^{n+1} = A(TG_P')^n + 1 \cdot [1 \cdot A(t_{1,n}) + A(s_{2,n+1})] + 0 \cdot [...]$$
$$= A(TG_P')^n + A(t_{2,n}) + A(s_{1,n+1})$$

Der Nachweis für Punkt 2 kann analog zum Induktionsschluß geführt werden.

qed.

Sei $MP = \{P_1, ..., P_m\}$ die Menge aller durch einen Transformationsgraphen gegebenen Programmvarianten. Dann erfüllt der Zeitaufwand $A(P_{eff})$ der effizientesten, diversitären Programmvariante der durch den Transformationsgraphen gegebenen Varianten folgende Gleichung:

$$A(P_{eff}) = \min (A(P_1), ..., A(P_m)).$$

Der Algorithmus zur Auswahl der für die Transformation der einzelnen Befehle eines Programms zu verwendenden Komplementart läßt sich also auf ein graphentheoretisches Problem zurückführen. Wenn jeder Pfeil des Transformationsgraphen mit der Ausführungsdauer gewichtet wird, dann lautet das durch den Algorithmus zu lösende Problem: "Suche den kürzesten Weg vom Startknoten E zum Zielknoten A". Die Bestimmung von "kürzesten Wegen" in gewichteten Graphen ist ein bekanntes und gut untersuchtes Problem, für das schon viele Algorithmen entwickelt wurden [/Nolt 76/, /Mehl 84/].

4 Ergebnisse

Aufbauend auf den Untersuchungen von /Hinz 89/ zur Hardwarefehlererkennung durch diversitäre Software wurden die nicht erkannten Fehler analysiert, die sich bei Verwendung von reiner Entwurfsdiversität ergeben haben. Bei diesen Untersuchungen wurden für eine Problemstellung jeweils zwei diversitäre Algorithmen verwendet, die in der Programmiersprache C implementiert waren. Eine Ursache für eine große Zahl nicht erkannter Fehler lag dabei in der Verwendung standardmäßig gegebener Bibliotheksroutinen. Diese Routinen entsprechen einer großen Anzahl von Befehlssequenzen, die bei den einzelnen Programmvarianten identisch ablaufen und damit zu größeren, nicht diversitären Programmteilen führen. Um solche gleichen Programmteile vermeiden zu können, wurden entwurfsdiversitäre Bibliotheksroutinen implementiert, die zu einer verbesserten Fehlererfassung auf Grund erhöhter Entwurfsdiversität führten.

Die in Bild 4.1 angegebenen Diagramme zeigen die Mittelwerte, die sich bei den entwurfsdiversitären Programmbeispielen ergeben haben.

Das Auftreten des gefährlich fehlerhaften Zustandes, bei dem ein falsches Ergebnis berechnet und durch Absolut- und Relativtest nicht als falsch erkannt wird, konnte bei den untersuchten Programmbeispielen durch die Verwendung von entwurfsdiversitären Programmvarianten reduziert, aber nicht ausgeschlossen werden.

Um das Auftreten des gefährlich fehlerhaften Zustandes weiter zu reduzieren, wird zusätzlich zur Entwurfsdiversität die beschriebene systematisch erzeugte Diversität eingesetzt. Für die Auswertung der erreichten Fehlerüberdeckung wurde eine Programmvariante der bisher verwendeten Programmbeispiele unverändert übernommen. Die zweite Variante wurde aus der anderen entwurfsdiversitären Variante mit dem Precompiler systematisch diversitär erzeugt. Anschließend wurde die Fehlererfassung bei den sowohl entwurfsdiversitären als auch systematisch diversitären Programmvarianten mit Hilfe des softwareimplementierten Fehlerinjektors untersucht, der die modellierten Fehlereffekte durch die Programmausführung im Einzelschrittmodus für jeden Programmbefehl erzeugen kann.

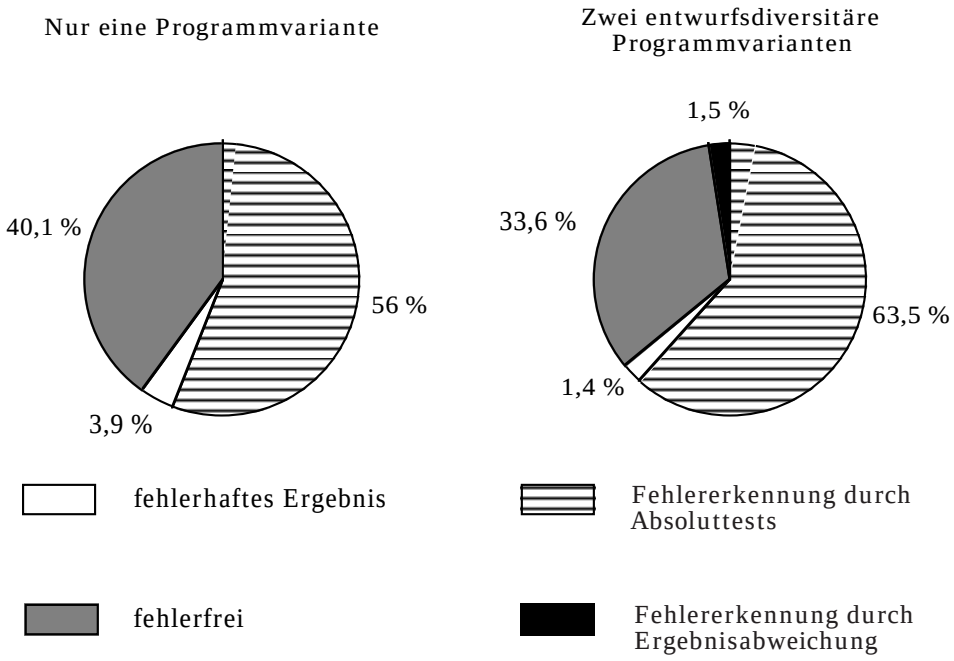

Bild 4.1: Vergleich der Ergebnisklassen zwischen einem Simplexsystem und einem virtuellen Duplexsystem mit entwurfsdiversitären Programmvarianten.

Der Precompiler zur systematischen Generierung von diversitären Programmvarianten wurde in einem Prototypen realisiert, in dem noch nicht alle Optimierungsmaßnahmen implementiert sind. Deshalb wurde die Fehlererfasssung bisher nur für zwei kleine Programmbeispiele untersucht, die von Hand optimiert wurden. Die Untersuchung der Fehlererfassung ergab, daß bei diesen Beispielen durch den Einsatz von systematisch erzeugter Diversität das Auftreten des gefährlich fehlerhaften Zustands um weitere 75% reduziert werden konnte. Da aus den Ergebnissen zweier kleiner Programmbeispiele noch keine statistischen Aussagen gemacht werden können, werden nach Fertigstellung des Precompilers umfangreichere Untersuchungen bezüglich der Fehlererfassung von sowohl entwurfsdiversitären als auch systematisch diversitären Programmvarianten durchgeführt.

5 Literatur

/ EHNi 90/ K. Echtle, B. Hinz, T. Nikolov: On Hardware fault detection by diverse software; 13th International conference on fault-tolerant systems and diagnostics, conf. proc., Verlag der Bulgarischen Akademie der Wissenschaften, 1990, S. 362 - 367.

/HaGö 91/ W. Hahn, M. Gössel: Pseudoduplication of floating point addition - a method of compiler generated checking of permanent hardware faults, Third European Workshop on Dependable Computing EWDC-3, Munich, April 1991

/Hinz 89/ B. Hinz: Erkennung von Mikroprozessor-Hardware-Fehlern mittels diversitär entwickelter Software; Diplomarbeit, Fakultät für Informatik, Univ. Karlsruhe, 1989.

/Mehl 84/ K. Mehlhorn: Datastructures and Algorithms 2 - Graph Algorithms and NP-Completeness, Springer Verlag, 1984.

/Mula 85/ M. Mulazzani: Reliability Versus Safety; Safecomp ´85, conf. proc., 1985, S. 141 - 146

/Nolt 76/ H. Noltemeier: Graphentheorie mit Algorithmen und Anwendungen, de Gruyter, Berlin, 1976.