

Verlässliche Echtzeitsysteme

Härtung von Daten und Kontrollfluss

Fabian Scheler

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

19. Juni 2012



1 Überblick

2 Arithmetisches Kodierung

- AN-Kodierung
- ANBD-Kodierung
- Arithmetische Kodierung des Kontrollflusses
- Implementierungen der ANBD-Kodierung

3 Combined Redundancy

4 Zusammenfassung



- vergangene Woche: „grob-granulare“ Redundanz
 - auf Ebene **kompletter Rechenknoten** und Prozessen
 - ↪ zum Zweck der **Fehlermaskierung**
 - durch **einfache Replikation** im Falle von „fail-silent“-Verhalten
 - durch **Mehrheitsentscheid** falls **Fehler im Wertbereich** auftreten



Heute: „fein-granulare“ Redundanz

- auf der Ebene **einzelner Instruktionen** und Datenelemente
- ↪ zum Zweck der **Fehlererkennung** und **-maskierung**
 - ↪ Implementierung von „fail-silent“-Verhalten
- **arithmetische Kodierung** von Werten und Berechnungen
 - Nutzung räumlicher Redundanz



kombinierter Einsatz grob- und fein-granularer Redundanz

- Ergänzung der Stärken, Eliminierung der Schwächen



1 Überblick

2 Arithmetisches Kodierung

- AN-Kodierung
- ANBD-Kodierung
- Arithmetische Kodierung des Kontrollflusses
- Implementierungen der ANBD-Kodierung

3 Combined Redundancy

4 Zusammenfassung



- Beispiel: Summation von drei Funktionsargumenten

```
int sum(int a, int b, int c) {  
    int result = a + b;  
    result = result + c;  
  
    return result;  
}
```

- Was kann hier schief alles gehen?
 - unter der Voraussetzung, dass das Programm korrekt ist
 - ~ es erzeugt also keine der Additionen einen Ganzzahlüberlauf



transiente Fehler können folgende Fehler hervorrufen

1 Operandenfehler

- der Wert des Operanden wird verfälscht oder ist veraltet
- der Operand selbst wird verfälscht ~ falsche(s) Speicherstelle/Register

2 Berechnungsfehler

- die Operation erzeugt ein falsches Ergebnis

3 Operatorfehler

- der Programmzähler/die Instruktion wird verfälscht ~ falschen Instruktion
- ~ Ausführung einer



Lösungsansatz: arithmetische Kodierung

Als Alternative oder Ergänzung zur redundanten Ausführung

- **Ausgangspunkt:** Kodierung von Nachrichten mithilfe von n Bits
- 👉 **Ansatz:** hinzufügen von k Prüfbits führt zu **räumlicher Redundanz**
 - ↪ es gibt 2^n gültige Nachrichten und insgesamt 2^{n+k} Worte
- die **Wahrscheinlichkeit** p_u , für einen **nicht entdeckten Fehler** ist:
 - der Fehler überführt also eine gültige wieder in eine gültige Nachricht

$$p_u = \frac{\text{Anzahl gültiger Nachrichten}}{\text{Anzahl möglicher Worte}} \approx \frac{2^n}{2^{n+k}} = 2^{-k}$$

- sofern man eine Gleichverteilung der Fehler zugrunde legt
- ↪ die Stärke der Absicherung hängt direkt an der Zahl k redundanter Bits
- betrachtet man die komplette Programmausführung, bedeutet dies:

$$p_u(x) = \left(1 - \frac{1}{2^k}\right)^{m-x} \left(\frac{1}{2^k}\right)^x \binom{m}{x}$$

- von insgesamt m **Instruktionen** sind also x **fehlerhaft**
 - ↪ Diese werden **nicht bemerkt**, es sind gültige Instruktionen (Nachrichten)

- für die Integration der Prüfbits gibt es verschiedene Möglichkeiten
systematisch eine Nachricht besteht aus n Bits
 - hiervon tragen k Bits die funktionale Bedeutung und
 - die übrigen $n - k$ Bits sind Prüfbits
 - hier ist ein direkter Zugriff auf den funktionalen Anteil möglich
 - eine spezielle Dekodierung ist nicht notwendig

separiert eine Nachricht ist ein 2er-Tupel, bestehend aus

- den k funktionalen Bits und $n - k$ Prüfbits
 - **getrennte Berechnung** des funktionalen Anteils und der Prüfbits
 - **nicht-separierte Codes** berechnen beides mit **derselben Operation**
- separierte Codes sind immer systematisch



systematische, nicht-separierte Kodierung ist attraktiv

- Behandlung des funktionalen Anteils/der Prüfbits in derselben Operation
- keine Dekodierung beim Zugriff auf den funktionalen Anteil



- die AN-Kodierung ordnet jeder Nachricht x eine kodierte x_c zu:

$$x_c = Ax; \quad A > 1$$

- kodierte Nachrichten sind also immer Vielfache von A
 - ein nicht-entdeckter Fehler müsste also wieder ein Vielfaches von A erzeugen
↪ Absicherung gegen Fehler im Wertbereich
- zurück kommt man durch **Modulo-Operation** und **Ganzzahldivision**

$$x_c \bmod A = 0 \quad x = x_c/A$$

- **Modulo-Operation** prüft die Korrektheit der Nachricht
 - **Ganzzahldivision** extrahiert den funktionalen Teil von x_c :
- ↪ die AN-Kodierung ist also **nicht-systematisch** und **nicht-separiert**

- die Nachricht wird durch k Bits gesichert: $A < 2^k$
 - ↪ den besten Schutz bieten also fast 2^k **große Primzahlen**
 - so ist die Wahrscheinlichkeit „**zufälliger Vielfacher**“ am geringsten



AN-Kodierte Operatoren

- die Kodierung eines Programms erfolgt vor dessen Laufzeit
 - zur Laufzeit muss also keine Primzahl A gefunden werden
 - **Konstanten** können **während der Übersetzung** kodiert werden
 - **Eingangsdaten** werden beim **Eintritt in das Programm** explizit kodiert
- ↪ im Programm selbst wird **nur mit kodierten Werten** gearbeitet



für jede Rechenoperation \circ ist eine kodierter Operator \circ_c nötig

- diese muss sowohl die Prüfbits als auch den funktionalen Teil x umfassen
 - Rechenoperationen finden immer **direkt auf den kodierten Werten statt**
 - des en- und dekodieren hinterließe den „**nackten, verwundbare Wert**“ x
- **kodierte Operatoren für grundlegende Arithmetik**

Operation	kodierter Op.	Implementierung	Bedeutung
Addition	$z_c = x_c +_c y_c$	$Az = Ax + Ay$	$A(x + y)$
Subtraktion	$z_c = x_c -_c y_c$	$Az = Ax - Ay$	$A(x - y)$
Multiplikation	$z_c = x_c *_c y_c$	$Az = (Ax * Ay)/A$	$A(x * y)$
Division	$z_c = \lfloor x_c /_c y_c \rfloor$	$Az = \lfloor (A * Ax) / Ay \rfloor$	$A \lfloor x/y \rfloor$



- **Beachte:** Die Operation erfolgt immer auf kodierten Werten!
 - Beispiel: Multiplikation $Az = (Ax * Ay)/A$
 - zuerst wird $Ax * Ay$ bestimmt
 - dann wird durch A dividiert
 - Gründe: würde man A sofort kürzen $\rightsquigarrow A(x * y)$
 - lägen wieder die „**nackten, verwundbaren Werte**“ x und y offen
 - die Operation **kennt x und y nicht**, nur die kodierte Nachrichten Ax und Ay
- **Beachte:** Multiplikation und Division benötigen **Korrekturen**
 - erfordern zusätzliche Multiplikation bzw. Division mit bzw. durch A
 - Addition und Multiplikation kommen hingegen ohne Korrektur aus

\rightsquigarrow Korrekturen sind potentiell immer **teure Operationen**
- **Beachte:** die kodierte Operatoren sind nur Implementierungsskizzen
 - sie sind nur aus mathematischer Sicht korrekt
 - sie beachten aber keine Feinheiten wie Über- oder Unterlauf



■ Operationen der booleschen Aussagenlogik

Operation	kodierter Op.	Implementierung	Bedeutung
Oder	$z_c = x_c \parallel_c y_c$	$z_c = x_c +_c y_c -_c x_c * y_c$	$A(x \parallel y)$
Und	$z_c = x_c \&\&_c y_c$	$z_c = x_c *_c y_c$	$A(x * y)$
Negation	$z_c = !_c x_c$	$z_c = 1_c -_c x_c$	$A(1 - x)$

↪ diese einfachen Operationen erfordern bereits teure Multiplikation

■ verschiedene Operatoren können nicht direkt kodiert werden:

- **Schiebeoperationen:** $x_c \ll_c y_c$ und $x_c \gg_c y_c$
- **bitweise boolesche Operatoren:** $x_c |_c y_c$, $x_c \&\&_c y_c$ und $\sim_c x_c$
- **Fließkommaarithmetik:** erfordert **Softwareemulation**
 - getrennte Behandlung von Vorzeichen, Exponent und Mantisse
 - können jeweils auf Ganzzahlarithmetik abgebildet werden

↪ auch hier werden **teure Berechnungsverfahren** nötig

- diese greifen auf die kodierten Standardoperatoren zu



- die AN-Kodierung deckt Fehler im Wertebereich ab
- manche Fehler führen aber dennoch zu einer gültigen Nachricht
 - **Operandenfehler** \rightsquigarrow Verwendung eines falschen Operanden
 - falls z. B. die Adresse beim Laden einer Speicherstelle verfälscht wird
 - die Operation läuft korrekt ab, auch das Ergebnis ist prinzipiell richtig \rightsquigarrow es wird aber der **semantisch falsche Wert** berechnet
 - **Operatorfehler** \rightsquigarrow Verwendung des falschen Operators
 - falls z. B. beim Laden der Operation ein Bit verfälscht wird
 - auch hier läuft die Operation korrekt ab \rightsquigarrow auch hier wird aber der **semantisch falsche Wert** berechnet



Erweiterung der Prüfbits

- sie sollen mehr semantische Informationen umfassen
 - Welche Operanden gehen in die Operation ein?
 - Welcher Operator ist für die Berechnung vorgesehen?

\rightsquigarrow ANBD-Kodierung



The Vital Coded Processor (VCP, [2])

- ursprünglich: ein durch ANBD-Kodierung geschützter Prozessor
 - teilweise werden Elemente **direkt in Hardware** implementiert
 - En- bzw. Dekodieren der ursprünglichen bzw. kodierten Nachricht
 - Überprüfung der Nachrichten und entsprechende Ausgangssteuerung
 - basierend auf dem Motorola 68000, später dem Motorola 68020
 - **kodierte Operationen** wurden **in Software** umgesetzt
 - Einsatz in **automatischen und halb-automatischen Zugführungssystemen**
 - Paris, Linie „RER A“, System „SACEM“
 - Lyon, Metrolinie „D“, System „MAGGALY“
 - Chicago, Flughafen, System „VAL“
- Erweiterung der AN-Kodierung um **statische Signaturen**:

$$x_c = Ax + B_x; \quad A > 1 \wedge B_x < A$$

- die Signatur B_x ist spezifisch für die Variable x_c
 - sie wird durch eine **statische Analyse** vorab bestimmt
 - der Quelltext der zu schützenden Anwendung muss bekannt sein



■ Fehlerüberprüfung und Dekodierung

$$x_c \bmod A = B_x \quad x = x_c/A$$

- Fehlerüberprüfung erfolgt durch Modulo-Operation
 - das Ergebnis ist jedoch die statische Signatur von x_c
- eine Ganzzahldivision extrahiert auch hier den Wert x
- Addition: $z_c = x_c +_c y_c = A(x + y) + B_x + B_y = A(x + y) + B_z$
 - die Signatur $B_z = B_x + B_y$ von z_c hängt von x_c und y_c ab
 - Signaturen für Eingangswerte werden zur Übersetzungszeit bestimmt
 - Signaturen für berechnete Werte werden daraus abgeleitet
 - auch hier muss gelten: $B_z = B_x + B_y < A$
- die Signatur von Berechnungsergebnisse ist abhängig von
 - der Signatur der Operanden \rightsquigarrow Eingabe für deren Bestimmung
 - der durchgeführten Operation \rightsquigarrow ihre Bestimmung selbst
 - wie die AN-Kodierung ist auch die ANBD-Kodierung nicht-separiert
 - die Signatur B_z wird direkt bei der Addition $x_c +_c y_c$ bestimmt



■ Beispiel: kodierte Summation dreier Summanden

```
1 int sum(int a_c, int b_c, int c_c) {
2     int result_c = a_c + b_c;
3     result_c = result_c + c_c;
4
5     return result;
6 }
```

- Berechnungsergebnisse und entsprechende Signaturen

Zeile 2 $a_c + b_c = A(a + b) + B_a + B_b$

Zeile 3 $a_c + b_c + c_c = A(a + b + c) + B_a + B_b + B_c$

■ angenommen es würden folgende Fehler auftreten:

- statt a_c wird x_c verwendet
 - ↪ die Signatur würde sich ändern: $B_x + B_b + B_c$
 - ↪ eine Erkennung des Fehlers ist gewährleistet
- Subtraktion statt einer Addition in Zeile 3
 - ↪ die Signatur würde sich ändern: $B_a + B_b - B_c$
 - ↪ eine Erkennung des Fehlers ist gewährleistet



Operationen auf veralteten Daten

- wird eine Variable **nicht aktualisiert**, wird dies bisher nicht erkannt
 - die Berechnung findet also mit veralteten Daten statt

☞ das „Alter“ eines Datums wird durch einen **Zeitstempel D** gesichert

$$x_c = Ax + B_x + D; \quad A > 1 \wedge B_x + D < A$$

- dieser Zeitstempel überwacht die **Anzahl der Variablenaktualisierungen**
 - der Zeitstempel muss **dynamisch zur Laufzeit** bestimmt werden
 - für die Überprüfung der Nachricht muss der erwartete Wert bekannt sein
- die Signatur B_x und A werden aber auch hier **statisch** bestimmt

☞ alle auf Folie 5 angenommen Fehler werden abgedeckt

- Operandenfehler, Operationsfehler und Operatorfehler



- keine direkte Kodierung der Division
 - Emulation durch wiederholte Subtraktion oder Rückfall zur AN-Kodierung
 - Addition, Subtraktion und Multiplikation werden unterstützt
- mehr aufwendige Korrekturoperationen sind erforderlich
 - für die Multiplikation gilt beispielsweise

$$\begin{aligned}x_c * c y_c &\neq A * x * y + B_x * B_y \\ &= A^2 * x * y + A * x * B_y + A * y * B_x + B_x * B_y\end{aligned}$$

- Was passiert eigentlich bei Fehlern im Kontrollfluss?
 - der falsche Grundblock im Kontrollflussgraphen wird angesprungen
 - weil z. B. die Entscheidung eines bedingten Sprungs verfälscht wird
 - einige Instruktionen werden übersprungen
 - weil z. B. der Instruktionszähler (engl. *program counter*) verfälscht wird



- Idee: auch der Grundblock x bekommen eine Signatur BB_x
 - BB_x umfasst die Summe aller im Grundblock x bestimmten Signaturen
- ☞ dies Signatur wird zur Laufzeit durch einen „Watchdog“ überwacht
 - er besitzt ein Feld s der zu erwartenden Signaturen BB_x
 - die Anwendung teilt den dynamisch bestimmten Wert für BB_x mit
- ☞ die Anwendung enthält eine Zählvariable acc
 - die sie zur Bestimmung von BB_x verwendet
 - Wert am Beginn des Grundblocks: $acc = s[i] - BB_x - x_{id}$
 - $s[i]$ enthält den erwarteten Wert nach dem Grundblock x
 - die statisch bestimmte Signatur BB_x wird abgezogen
 - ebenso eine eindeutige ID $x_{id} \rightsquigarrow$ bedingte Sprünge (s. Folie VIII/20)
 - acc wird kontinuierlich um die jeweils bestimmte Signatur inkrementiert
 - für den nachfolgenden Grundblock wird acc neu initialisiert
 - hierfür speichert das kodierte Programm ein Feld $delta[i] = s[i + 1] - s[i]$
 - am Ende eines Grundblocks wird dieser Wert addiert $\rightsquigarrow acc = s[i + 1] - x_{id}$
 - dann werden Signatur bzw. ID addiert/subtrahiert $\rightsquigarrow acc = s[i] - BB_y - y_{id}$



■ unkodierter Grundblock:

- zur Vereinfachung direkt in einer maschinencodeähnlichen Darstellung

```
1 bb1 :  
2   x = a + b           - eine Addition gefolgt von einer Subtraktion  
3   y = x - d           - unbedingter Sprung zu einem weiteren Grundblock  
4   br bb2
```

■ Kodierung des Grundblocks:

```
1 bb1 :  
2   x_c = a_c + b_c  
3   acc += x_c % A  
4   y_c = x_c - d_c  
5   acc += y_c % A  
6  
7   send(acc, bb1_id)  
8   acc += delta[i]  
9   i++  
10  acc += bb1_id  
11  acc -= BB_b2  
12  acc -= bb2_id  
13  br bb2
```

- 1** Kodierung der Berechnungen
- 2** Bestimmung der Signatur BB_{bb1} in acc
 - zu Beginn gilt: $acc = s[i] - BB_{bb1} - bb1_{id}$
 - Zeile 3: $acc = s[i] - BB_{bb1} - bb1_{id} + B_a + B_d$
 - Zeile 5: $acc = s[i] - bb1_{id}$
- 3** Signatur an den „Watchdog“ senden (Zeile 7)
- 4** Vorbereitungen für den Grundblock bb2
 - Zeile 8: $acc = s[i + 1] - bb1_{id}$
 - Zeile 12: $acc = s[i] - BB_{bb2} - bb2_{id}$



Kodierung bedingter Sprünge [3]

- **Herausforderung:** Übertragung des Konzepts für sequentiellen Code
~> Wie funktioniert hier die Umschaltung zwischen Grundblöcken?
 - Welcher der nächste Grundblock ist, hängt ja vom bedingten Sprung ab ...

- **Außerdem:** es gibt neue Möglichkeiten für Fehler
 - das **Ergebnis der Entscheidung** könnte verfälscht werden
 - der **bedingte Sprung** selbst könnte verfälscht werden

☞ man muss das Ergebnis der Entscheidung absichern

~> hier hilft es, dieses Ergebnis arithmetisch zu kodieren

☞ man muss sicherstellen, dass der bedingte Sprung korrekt ist

~> hier helfen die IDs der angesprungenen Grundblöcke

- sind vorab bekannt ~> geben an, in welchem Grundblock man sein muss

- **unkodierter Grundblock:**

```
1 bb1 :  
2   cond = ...           - cond speichert die Sprungentscheidung  
3   br cond bbt bbf     - br springt dann zu bbt (wahr) oder bbf (falsch)
```



Kodierung bedingter Sprünge (Forts.)

Wie sieht das aus? Erläuterung anhand eines Beispiels aus [3].

```
1 bb1:
2   cond_c = ...
3   acc += cond_c % A
4   send(acc, bb1_id)
5
6   acc += delta[i]
7   i++
8   acc += bb1_id - BB_bbt - bbt_id - (A * 1 + B_cond)
9
10  cond = cond_c % A
11  acc += cond_c
12  br cond bbt bbf_cor
```

1 anfangs: $acc = s[i] - BB_{bb1} - bb1_{id}$

2 Zeile 2: die Bedingung wird kodiert $\rightsquigarrow cond_c$

- wahr $\rightsquigarrow cond_c = A * 1 + B_{cond}$ und falsch $\rightsquigarrow A * 0 + B_{cond}$

3 Zeile 4: sende $acc = s[i] - BB_{bb1} - bb1_{id} + B_{cond}$ an den „Watchdog“

4 Zeile 6-8: bereite acc für den Sprung auf bbt vor

\rightsquigarrow nun gilt $acc = s[i] - BB_{bbt} - bbt_{id} - (A * 1 + B_{cond})$

5 Zeile 10-12: extrahiere Wert von $cond_c$ \rightsquigarrow aktualisiere acc und springe

\rightsquigarrow nun gilt $acc = s[i] - BB_{bbt} - bbt_{id} - (A * 1 + B_{cond}) + cond_c$



Kodierung bedingter Sprünge (Forts.)

Wie sieht das aus? Erläuterung anhand eines Beispiels aus [3].

```
1 bb1:                                1 bbt:
2   cond_c = ...                       2   ...
3   acc += cond_c % A                   3
4   send(acc, bb1_id)                   4 bbf_cor:
5                                         5   acc += A
6   acc += delta[i]                     6   acc += BB_bbt + bbt_id
7   i++                                  7   acc -= BB_bbf - bbf_id
8   acc += bb1_id - ...                  8   br bbf
9                                         9
10  cond = cond_c % A                    10 bbf:
11  acc += cond_c                         11   ...
12  br cond bbt bbf_cor
```

6 für $cond = \text{wahr}$ bleibt nichts zu tun, schließlich gilt $cond_c = A * 1 + B_{cond}$

↪ insgesamt gilt: $acc = s[i] - BB_{bbt} - bbt_{id}$, der Anfangswert für den Grundblock bbt

7 für einen Sprung zu bbf ist aber eine Korrektur notwendig

- schließlich wurde acc für einen Sprung zu bbt vorbereitet

8 Zeile 4: eingangs gilt $acc = s[i] - BB_{bbt} - bbt_{id} - A * 1$

- hier gilt $cond_c = A * 0 + B_{cond} = B_{cond}$

↪ korrigiert: $acc = s[i] - BB_{bbf} - bbf_{id}$, der Anfangswert für des Grundblocks bbf

9 nun kann weiter zu bbf gesprungen werden



- interpretiert binäre Maschinencodeabbilder eines Programms
 - Zielsystem ist der DLX-Prozessor
 - ein RISC-Prozessor für akademische Anwendungsgebiete
 - Konstanten, Speicheradressen etc. werden zur Ladezeit kodiert
 - kodierte Operationen sind in Software implementiert



Fehlerinjektion \rightsquigarrow Fehlererkennungsrate ist sehr gut

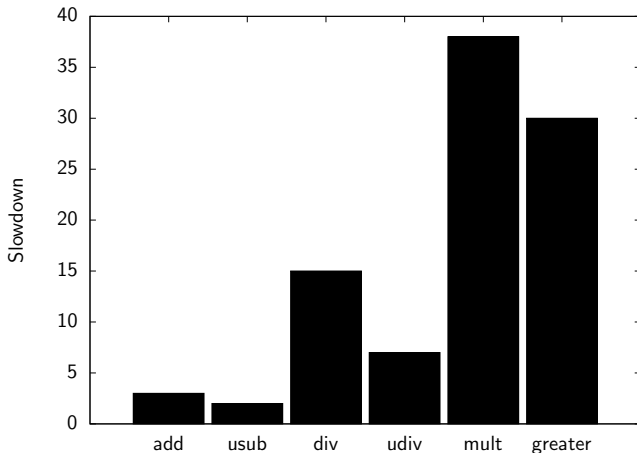
- kodierter Interpreter: keine fehlerhaften Ergebnisse
 - nicht-kodierte Ausführung:
 - interpretiert: 4% der Ergebnisse fehlerhaft
 - native Ausführung: 9% der Ergebnisse fehlerhaft
- \rightsquigarrow Interpreter verdeckt bereits diverse Fehler



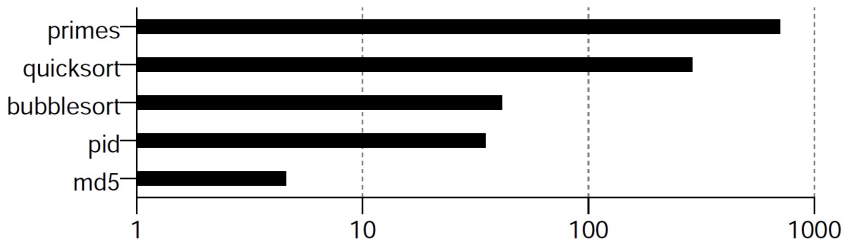


sehr hohe Laufzeitkosten interpretierter kodierter Operationen

- im Vergleich zu interpretierten aber nicht-kodierten Operationen
- eine Multiplikation dauert 38-mal so lange ...



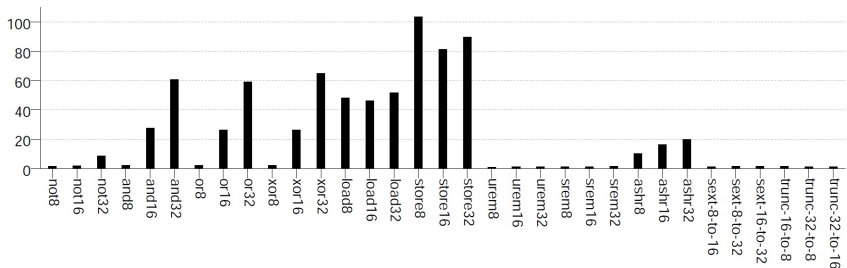
- Kodierung wird **vor der Laufzeit durch einen Compiler** durchgeführt
 - nicht mehr zur Laufzeit durch einen Interpreter
- ☞ hierfür muss aber der **Quelltext** vorhanden sein
 - Nur in **Binärform** vorliegende Bibliotheken stellen ein Problem dar!
- ↪ hier kommen **Hüllfunktionen** (engl. *wrapper*) zum Einsatz
 - diese extrahieren die eigentlichen Werte der kodierten Variablen
 - die Berechnung selbst findet dann nicht-kodiert also ungeschützt statt
- ☞ allerdings sind die Geschwindigkeitszugewinne beträchtlich:
 - Beschleunigung im Vergleich zum interpretierenden SEP



- Vergleich mit nativ ausgeführten Operationen

↪ fördert die **wahren Laufzeitkosten** zutage

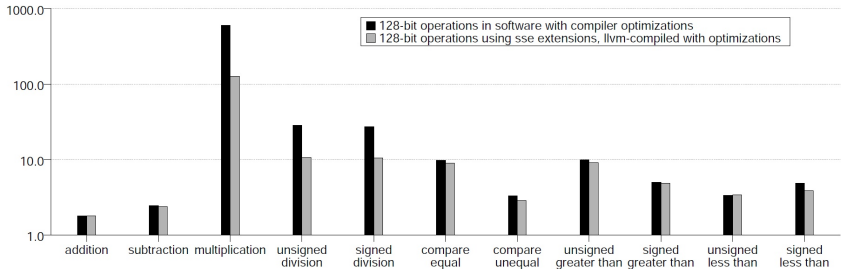
- Operationen, die nicht direkt kodierbar sind:



- das Speichern eines 8 Bit großen Wortes ist bis zu 100x langsamer
 - diese Operation besteht aus diversen Einzelschritten
 - ↪ Laden, bitweises Und, Schiebeoperation, ...
 - ↪ all das muss in kodierter Form ablaufen, all das ist teuer



■ direkt kodierbare arithmetische Operationen



- auch hier sind Laufzeitkosten zum Teil beträchtlich
 - Addition und Subtraktion sind vergleichsweise günstig
 - einfache Vergleichsoperationen sind aber relativ teuer
- einzig Multiplikation und Division benötigen 128-bit Operationen
 - sie profitieren aber enorm von den SSE-Erweiterungen heutiger Prozessoren



1 Überblick

2 Arithmetisches Kodierung

- AN-Kodierung
- ANBD-Kodierung
- Arithmetische Kodierung des Kontrollflusses
- Implementierungen der ANBD-Kodierung

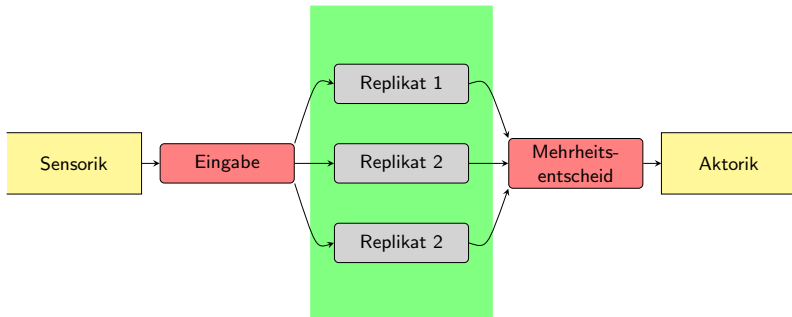
3 Combined Redundancy

4 Zusammenfassung



Combined Redundancy (CoRed, [4])

- die kodierte Ausführung kompletter Programme ist derzeit **zu teuer**
ihre **selektiver Einsatz** erscheint hingegen vielversprechend
- ↪ verbliebene Bruchstellen von TMR (s. Folie VIII/22) abzusichern



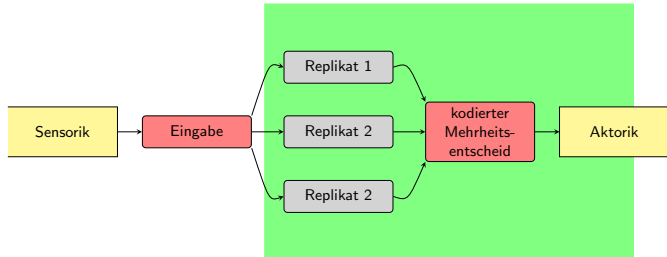
- diese verbliebenen kritischen Schwachpunkte sind
 - die Eingaben und die Einigung über diese Eingaben
 - und der Mehrheitsentscheid sowie die Weitergabe an die Aktoren

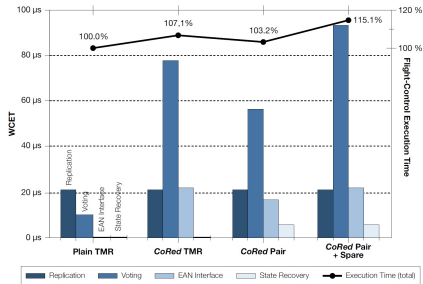




genau diesen Weg beschreibt CoRed

- die eigentlich **Berechnung wird durch Redundanz geschützt**
 - hier wäre eine arithmetische Kodierung zu teuer
- die **kritischen Bruchstellen werden arithmetisch kodiert**
 - dies umfasst verhältnismäßig wenig Code
 - ~> die Laufzeitkosten halten sich in Grenzen
- weitere redundante Rechenschritte sind hier nicht die optimale Lösung
 - d. h. redundante Einigungen und Mehrheitsentscheide
 - ~> sie reduzieren zwar die Fehlerwahrscheinlichkeit
 - ~> bringen aber immer weitere kritische Bruchstellen hervor





selektive Anwendung von arithmetischer Kodierung

→ sehr gute Fehlertoleranz

→ bei vertretbaren Kosten

- Balkengrafik gibt **nur die Mehrkosten** der einzelnen Komponenten an
 - also Mehrkosten für die replizierte Ausführung, Mehrheitsentscheid, ...
 - der Aufwand für den Mehrheitsentscheid steigt durch Kodierung enorm
 - das sind die Datensätze „Plain TMR“ und „CoRed TMR“
- die Kurve bezieht sich auf die **gesamte Ausführungszeit**
 - „CoRed TMR“ benötigt hier also nur 7,1% mehr Zeit als „Plain TMR“
 - würde man alles kodieren, wäre man hier bei mehreren 100%
 - die selektive Anwendung arithmetischer Kodierung bringt



1 Überblick

2 Arithmetisches Kodierung

- AN-Kodierung
- ANBD-Kodierung
- Arithmetische Kodierung des Kontrollflusses
- Implementierungen der ANBD-Kodierung

3 Combined Redundancy

4 Zusammenfassung



Fehlererkennung möglichst ohne redundante Ausführung

- Erkennung von Operanden-, Berechnungs- und Operatorfehlern
- ↪ Einsatz räumlicher Redundanz durch Prüfbits

arithmetisch Kodierung

- (nicht-)systematisch und (nicht-)separiert

AN-Kodierung ↪ Fehler im Wertbereich

- Kodierung: Multiplikation mit einem konstanten Faktor A
- kodierte Addition, Subtraktion, Multiplikation, Division
- Aussagenlogik, Schiebeoperatoren, Fließkommaarithmetik

ANBD-Kodierung erweitert die AN-Kodierung

- um statische Signaturen und dynamische Zeitstempel
- Kodierung des Kontrollflusses ↪ Signaturen für Grundblöcke

CoRed-Ansatz ↪ selektive Anwendung der ANBD-Kodierung

- durchgehende arithmetische Kodierung wäre zu teuer



- [1] FETZER, C. ; SCHIFFEL, U. ; SÜSSKRAUT, M. :
AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware.
In: BUTH, B. (Hrsg.) ; RABE, G. (Hrsg.) ; SEYFARHT, T. (Hrsg.): *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP '09)*.
Heidelberg, Germany : Springer-Verlag, 2009. –
ISBN 978-3-642-04467-0, S. 283-296
- [2] FORIN, P. :
Vital coded microprocessor principles and application for various transit systems.
In: *Selected Papers from the IFAC/IFIP/IFORS Symposium on Control, computers, communications in transportation*.
Oxford, UK : Pergamon Press, Sept. 1989. –
ISBN 008037025X, S. 79-84
- [3] SCHIFFEL, U. ; SCHMITT, A. ; SÜSSKRAUT, M. ; FETZER, C. :
ANB- and ANBDbmem-encoding: detecting hardware errors in software.
In: SCHOITSCH, E. (Hrsg.): *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security (SAFECOMP '10)*.
Heidelberg, Germany : Springer-Verlag, 2010. –
ISBN 978-3-642-15650-2, S. 169-182



- [4] ULBRICH, P. ; HOFFMANN, M. ; KAPITZA, R. ; LOHMANN, D. ;
SCHRÖDER-PREIKSCHAT, W. ; SCHMID, R. :
Eliminating Single Points of Failure in Software-Based Redundancy.
In: *Proceedings of the 9th European Dependable Computing Conference (EDCC '12)*,
IEEE Computer Society Press. –
(To appear)
- [5] WAPPLER, U. ; FETZER, C. :
Software Encoded Processing: Building Dependable Systems with Commodity
Hardware.
In: SAGLIETTI, F. (Hrsg.) ; OSTER, N. (Hrsg.): *Proceedings of the 26th International
Conference on Computer Safety, Reliability, and Security (SAFECOMP '07)*.
Heidelberg, Germany : Springer-Verlag, 2007. –
ISBN 978-3-540-75100-7, S. 356–369

