Verlässliche Echtzeitsysteme

Härtung von Daten und Kontrollfluss

Peter Ulbrich

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)

www4.informatik.uni-erlangen.de

05. Mai 2014

Fragestellungen

- 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

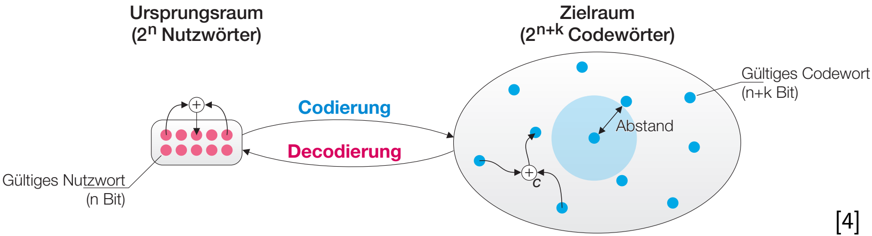
- auf der Ebene einzelner Instruktionen und Datenelemente
- ⇝ zum Zweck der Fehlererkennung und -maskierung
  - ⇝ Implementierung von „fail-silent“-Verhalten
- arithmetische Codierung von Werten und Berechnungen
  - Nutzung räumlicher Redundanz

☞ kombinierter Einsatz grob- und fein-granularer Redundanz

- Ergänzung der Stärken, Eliminierung der Schwächen

Codierung: Einsatz von Informationsredundanz

Als Alternative oder Ergänzung zur redundanten Ausführung



[4]

- Koordinierter Einsatz von Informationsredundanz ⇝ Codierung
- Ausgangspunkt: Darstellung der Nutzdaten mithilfe von n Bits

☞ Ansatz: Hinzufügen von k Prüfbits führt zu Informationsredundanz

⇝ weiterhin 2^n gültige Codeworte bei nunmehr 2^{n+k} möglichen Worten
- Überführung mittels Codierungsvorschrift
- Fehlererkennung ⇝ Absoluttest (Konformität mit Vorschrift)
- es genügt eine Instanz für die Fehlererkennung ≠ Replikation

Codierung: Restfehlerwahrscheinlichkeit

Fehler bleiben unentdeckt, wenn ...

- Schwere des Fehlers spielt eine Rolle ($\neq$ Replikation)
- Restfehlerwahrscheinlichkeit p_{sdc}, für unerkannte Datenfehler ist:
  - der Fehler überführt also eine gültige wieder in eine gültige Nachricht

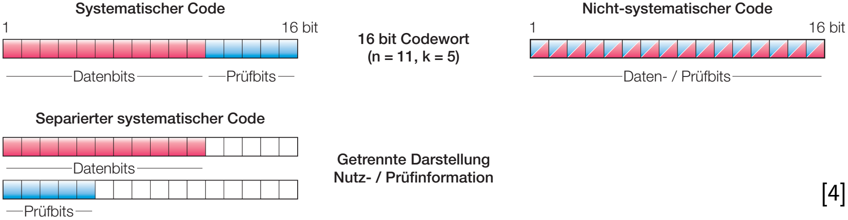
$$p_{sdc} = \frac{\text{Anzahl gültiger Nachrichten}}{\text{Anzahl möglicher Worte}} \approx \frac{2^n}{2^{n+k}} = 2^{-k}$$

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

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

  - von insgesamt m Instruktionen sind also x fehlerhaft
    - $\leadsto$ diese werden nicht bemerkt, es sind gültige Instruktionen (Nachrichten)

Codierung: Darstellung der Codewörter



[4]

- für die Integration der Prüfbits gibt es verschiedene Möglichkeiten
- systematischer vs. nicht-systematischer Code
  - Speicherstellen der n Daten- und k Prüfbits sind trennbar vs. vermischt
  - Zugriff auf die Nutzdaten ohne Decodierung ist möglich vs. nicht möglich
- separierter Code: 2er-Tupel
  - getrennte Berechnung des funktionalen Anteils und der Prüfbits
  - nicht-separierte Codes berechnen beides mit derselben Operation
  - separierte Codes sind immer systematisch
- systematische, nicht-separierte Codierung ist attraktiv
  - Behandlung des funktionalen Anteils/der Prüfbits in derselben Operation
  - keine Dekodierung beim Zugriff auf den funktionalen Anteil

Fehlermodell: Was kann alles schief gehen?

- 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 alles schief gehen?
    - unter der Voraussetzung, dass das Programm korrekt ist
    - ⇝ keine der Additionen erzeugt 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
   - ⇝ Ausführung einer falschen Instruktion

Gliederung

Arithmetische Codierung: Grundlegende AN-Codes

- arithmetische Codierung zum Schutz von Berechnungen geeignet
- die Codierung überführt den Wert v in einen codierten Wert v_c:

$$v_c = A \cdot v; \quad A > 1$$

  - codierte Werte sind also immer Vielfache von A
    - ein unerkannter Fehler müsste abermals ein Vielfaches von A erzeugen
    - $\rightsquigarrow$ Absicherung gegen Fehler im Wertbereich
  - zurück kommt man durch Modulo-Operation und Ganzzahldivision

$$v_c \mod A = 0 \quad v = v_c / A$$

    - Modulo-Operation prüft die Korrektheit der Nachricht
    - Ganzzahldivision extrahiert den funktionalen Teil von v_c
  - $\rightsquigarrow$ die AN-Codierung ist also nicht-systematisch und nicht-separiert

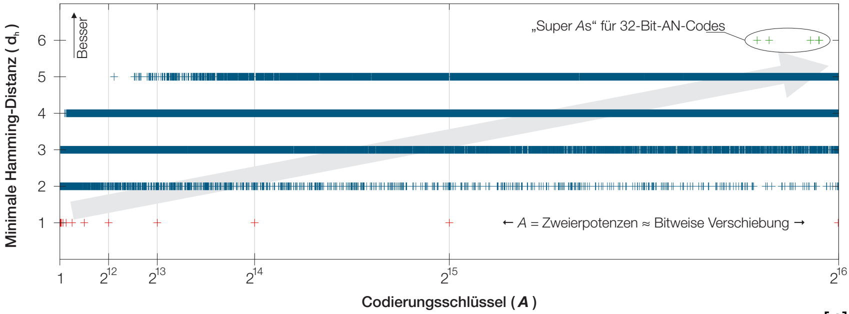
Restfehlerwahrscheinlichkeit: Wähle ein geeignetes A!

Die Binärdarstellung stellt besondere Anforderungen.

- Bitkipper erzeugen evtl. gültige Codewörter $\leadsto$ p_{sdc}
  - die Wahrscheinlichkeit hängt vom Abstand der Codewörter hab
  - ist jedoch nie Null
- der Codierungsschlüssel A bestimmt die Robustheit
  - aus mathematischer Sicht sinnvoll: große Primzahlen
  - Codierte Datenströme sollen möglichst Teilerfremd sein
- In der Praxis entscheidend: robuste Bitmuster
  - für binär-codierte Daten hängt dies von der Hammingdistanz d_h ab
  - $\leadsto$ an wievielen Bits unterscheiden sich zwei Nachrichten
  - Erfreuliche Eigenschaft: $d_h - 1$ Bitfehler werden sicher erkannt

Wähle ein geeignetes A! (Forts.)

Experimentelle Bestimmung der Hamming-Distanz



[4]

- betrachte alle gültigen Codewörter $A \cdot v \leadsto$ min. Hamming-Distanz
  - große Schwankungen $\leadsto$ größer ist nicht automatisch besser
  - Primzahlen sind gut, die Besten sind jedoch zusammengesetzte Zahlen

  $\leadsto$ Super As mit $d_h = 6$: 58659, 59665, 63157, 63859 und 63877
  - für 32-Bit-AN-Codes mit 16-Bit-Schlüsseln

AN-Codes: Operatoren

- die Codierung eines Programms erfolgt vor dessen Laufzeit
  - zur Laufzeit muss also keine Schlüssel A gefunden werden
  - Konstanten können während der Übersetzung codiert werden
  - Eingangsdaten werden beim Eintritt in das Programm explizit codiert
  - ⇝ im Programm selbst wird nur mit codierten Werten gearbeitet

☞ für jede Rechenoperation $\circ$ ist eine codierter Operator $\circ_c$ nötig

- diese muss sowohl die Prüfbits als auch den funktionalen Teil v umfassen
- Rechenoperationen finden immer direkt auf den codierten Werten statt
  - dessen en- und decodieren hinterließe den „nackten, verwundbare Wert" v

- codierte Operatoren für grundlegende Arithmetik

Operation	codierter 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	$z_c = x_c \cdot_c y_c$	$Az = (Ax \cdot Ay)/A$	$A(x \cdot y)$
Division	$z_c = \lfloor x_c /_c y_c \rfloor$	$Az = \lfloor (A \cdot Ax) / Ay \rfloor$	$A \lfloor x/y \rfloor$

AN-Kodierte Operatoren (Forts.)

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

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

Weitere Operationen

- Operationen der booleschen Aussagenlogik

Operation	codierter Op.	Implementierung	Bedeutung
Oder	$z_c = x_c \,\|\|_c\, y_c$	$z_c = x_c +_c y_c -_c x_c \cdot y_c$	$A(x \,\|\|\, y)$
Und	$z_c = x_c \,\&\&_c\, y_c$	$z_c = x_c \cdot_c y_c$	$A(x \cdot y)$
Negation	$z_c = !_c\, x_c$	$z_c = 1_c -_c x_c$	$A(1 - x)$

$\leadsto$ diese einfachen Operation erfordern bereits teure Multiplikation

- verschiedene Operatoren können nicht direkt codiert werden:
  - Schiebeoperationen: $x_c <<_c y_c$ und $x_c >>_c y_c$
  - 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

$\leadsto$ auch hier werden teure Berechnungsverfahren nötig

- diese greifen auf die codierten Standardoperatoren zu

Grenzen der AN-Codierung

- die AN-Codierung deckt Fehler im Wertebereich ab
- manche Fehler führen jedoch dennoch zu einem gültigen Codewort
  - Operandenfehler $\leadsto$ 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
    - $\leadsto$ es wird aber der semantisch falsche Wert berechnet
  - Operatorfehler $\leadsto$ Verwendung des falschen Operators
    - falls z. B. beim Laden der Operation ein Bit verfälscht wird
    - auch hier läuft die Operation korrekt ab
    - $\leadsto$ 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?

$\leadsto$ ANB-Codierung

ANB-Codes: Funktionsweise

- Erweiterung der AN-Codierung um statische Signaturen:

$$v_c = A \cdot v + B_v; \quad A > 1 \wedge B_v < A$$

  - die Signatur B_v ist spezifisch für die Variable v_c
    - sie wird durch eine statische Analyse vorab bestimmt
    - der Quelltext der zu schützenden Anwendung muss bekannt sein
  - Fehlerüberprüfung und Dekodierung

$$v_c \mod A = B_v \quad v = v_c / A$$

- 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: $B_z = B_x + By < A$
- 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-Codierung ist auch die ANB-Codierung nicht-separiert
    - die Signatur B_z wird direkt bei der Addition $x_c +_c y_c$ bestimmt

Fehlererkennung durch ANB-Codierung

- Beispiel: codierte 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
    - $\leadsto$ die Signatur würde sich ändern: $B_x + B_b + B_c$
    - $\leadsto$ eine Erkennung des Fehlers ist gewährleistet
  - Subtraktion statt einer Addition in Zeile 3
    - $\leadsto$ die Signatur würde sich ändern: $B_a + B_b - B_c$
    - $\leadsto$ eine Erkennung des Fehlers ist gewährleistet

The Vital Coded Processor (VCP, [2])

Bislang vollständigste Variante der arithmetschen Codierung

- Forin erweitert den Ansatz um Zeitstempel $D \rightsquigarrow$ veraltete Daten
- ursprünglich: ein durch ANBD-Codierung geschützter Prozessor
  - teilweise werden Elemente direkt in Hardware implementiert
    - En- bzw. Dekodieren der ursprünglichen bzw. codierten Nachricht
    - Überprüfung der Nachrichten und entsprechende Ausgangssteuerung
    - basierend auf dem Motorola 68000, später dem Motorola 68020
  - codierte 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“

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

$$v_c = A \cdot v + B_v + D; \quad A > 1 \wedge B_v + D < A$$

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

☞ alle auf Folie 6 angenommen Fehler werden abgedeckt

- Operandenfehler, Operationsfehler und Operatorfehler

Grenzen der ANBD-Codierung

- keine direkte Codierung der Division
  - Emulation durch wiederholte Subtraktion oder Rückfall zur AN-Codierung
  - Addition, Subtraktion und Multiplikation werden unterstützt

- mehr aufwendige Korrekturoperationen sind erforderlich
  - für die Multiplikation gilt beispielsweise

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

- 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

Direkte Codierung des Kontrollflusses nach Forin [2]

Requires: $B_x, B_y, B_{true}, B_{false}$ $\rightsquigarrow$ Konstante Signaturen für Operanden und Zweige
State: x_c, y_c, B_{cond}

1 **if** $(\textsc{decode}(x_c) \geq \textsc{decode}(y_c))$ **then** $B_{cond} \leftarrow B_{true}$ **else** $B_{cond} \leftarrow B_{false}$
2
3 **if** $(\textsc{decode}(x_c) \geq \textsc{decode}(y_c))$ **then**
4 $\quad y_c \leftarrow x_c - y_c$ $\rightsquigarrow$ Signatur: $B_x - B_y$
5 **else**
6 $\quad y_c \leftarrow x_c + y_c$ $\rightsquigarrow$ Signatur: $B_x + B_y$
7 $\quad y_c \leftarrow y_c - (B_x + B_y) + (B_x - B_y)$ $\rightsquigarrow$ Signaturanpassung: $B_x - B_y$
8 $\quad y_c \leftarrow y_c - B_{false} + B_{true}$ $\rightsquigarrow$ Verzweigung signieren
9 **end if**
10
11 $y_c \leftarrow y_c + B_{cond}$ $\rightsquigarrow$ Signaturanpassung, Sollwert: $B_x - B_y + B_{true}$

- Idee: Kolltrollflussabhängige Signaturanpassung
  - Ziel ist der Sollwert in Zeile 11 (`true`-Fall + B_{true})
  - Anpassung im `else`-Fall
- Nachteil: Gemeinsamer Operanden (hier: y_c) und Berechnungen in beiden Zweigen notwendig

Indirekte Codierung des Kontrollflusses [3]

- Idee: auch der Grundblock x bekommt eine Signatur BB_x
  - BB_x umfasst die Summe aller im Grundblock x bestimmten Signaturen
- ☞ die 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: $\texttt{acc} = \texttt{s[i]} - BB_x - x_{id}$
    - `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 V/23)
  - `acc` wird kontinuierlich um die jeweils bestimmte Signatur inkrementiert
  - für den nachfolgenden Grundblock wird `acc` neu initialisiert
    - hierfür speichert das codierte Programm ein Feld $\texttt{delta[i]} = \texttt{s[i+1]} - \texttt{s[i]}$
    - am Ende eines Grundblocks wird dieser Wert addiert $\rightsquigarrow \texttt{acc} = \texttt{s[i+1]} - x_{id}$
    - dann werden Signatur bzw. ID addiert/subtrahiert $\rightsquigarrow \texttt{acc} = \texttt{s[i]} - BB_y - y_{id}$

Codierung sequentieller Instruktionsfolgen

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

- uncodierter Grundblock:
  - zur Vereinfachung direkt in einer maschinencodeähnlichen Darstellung

```
bb1:
  x = a + b
  y = x - d
  br bb2
```

- eine Addition gefolgt von einer Subtraktion
- unbedingter Sprung zu einem weiteren Grundblock

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

Codierung bedingter Sprünge [3]

- Herausforderung: Übertragung des Konzepts für sequentiellen Code

$\leadsto$ 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

$\leadsto$ hier hilft es, dieses Ergebnis arithmetisch zu kodieren

☞ man muss sicherstellen, dass der bedingte Sprung korrekt ist

$\leadsto$ hier helfen die IDs der angesprungenen Grundblöcke
  - sind vorab bekannt $\leadsto$ geben an, in welchem Grundblock man sein muss

- uncodierter Grundblock:

```
bb1:
  cond = ...
  br cond bbt bbf
```

- cond speichert die Sprungentscheidung
- br springt dann zu bbt (wahr) oder bbf (falsch)

Codierung 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 $= \mathtt{s[i]} - BB_{bb1} - bb1_{id}$

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

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

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

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

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

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

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

Codierung 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 - ...
9
10   cond = cond_c % A
11   acc += cond_c
12   br cond bbt bbf_cor
```

```
1  bbt:
2    ...
3
4  bbf_cor:
5    acc += A
6    acc += BB_bbt + bbt_id
7    acc -= BB_bbf - bbf_id
8    br bbf
9
10 bbf:
11   ...
```

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

⇝ insgesamt gilt: acc $= \texttt{s[i]} - BB_{bbt} - bbt_{id}$, 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 $= \texttt{s[i]} - BB_{bbt} - bbt_{id} - A \cdot 1$

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

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

9 nun kann weiter zu bbf gesprungen werden

Software Encoded Processor (SEP, [6])

Programme codiert in einem Interpreter ausführen

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

☞ Fehlerinjektion $\leadsto$ Fehlererkennungsrate ist sehr gut

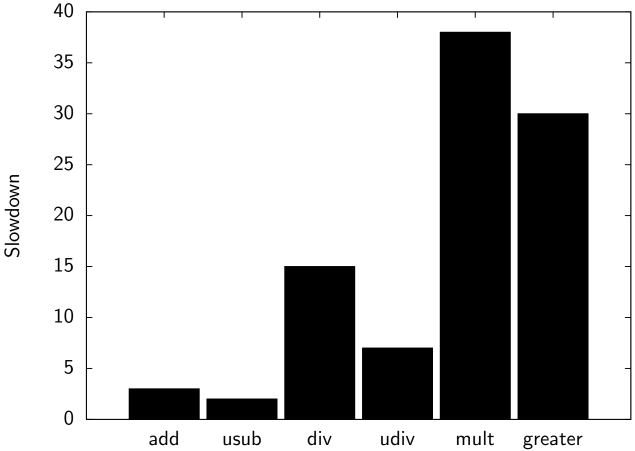
- codierter Interpreter: keine fehlerhaften Ergebnisse
- nicht-codierte Ausführung:
  - interpretiert: 4% der Ergebnisse fehlerhaft
  - native Ausführung: 9% der Ergebnisse fehlerhaft

  $\leadsto$ Interpreter verdeckt bereits diverse Fehler

Software Encoded Processor (SEP) (Forts.)

☞ sehr hohe Laufzeitkosten interpretierter codierter Operationen

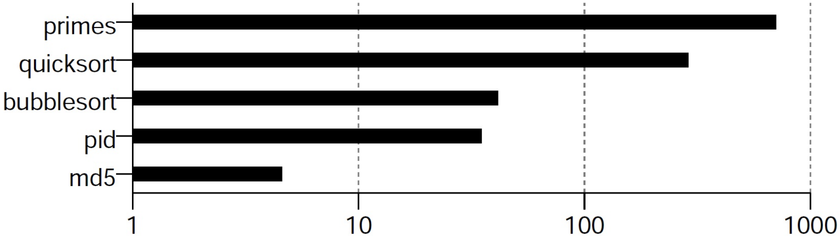
- im Vergleich zu interpretierten aber nicht-codierten Operationen
- eine Multiplikation dauert 38-mal so lange ...



Compiler Based Encoding (CBE, [3])

Quelle Grafik: [3]

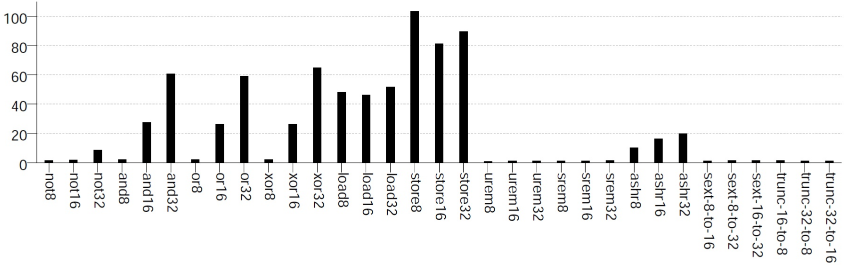
- Codierung wird vor der Laufzeit durch einen Compiler durchgeführt
  - nicht mehr zur Laufzeit durch einen Interpreter
- ☞ hierfür muss aber der Quelltext vorhanden sein
  - Nur in Binärform vorliegende Bibliotheken stellen ein Problem dar!
  - ⇝ hier kommen Hüllfunktionen (engl. *wrapper*) zum Einsatz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



Compiler Based Encoding (CBE) (Forts.)

Quelle Grafik: [1]

- Vergleich mit nativ ausgeführten Operationen
  - ⇝ fördert die wahren Laufzeitkosten zutage
- Operationen, die nicht direkt kodierbar sind:

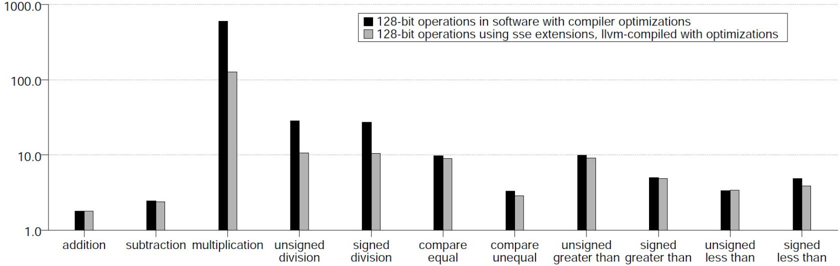


- das Speicher eines 8 Bit großen Wortes ist bis zu 100x langsamer
  - diese Operation besteht aus diversen Einzelschritten
  - ⇝ Laden, bitweises Und, Schiebeoperation, ...
  - ⇝ alle das muss in codierter Form ablaufen, all das ist teuer

Compiler Based Encoding (CBE) (Forts.)

Quelle Grafik: [1]

- 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

Gliederung

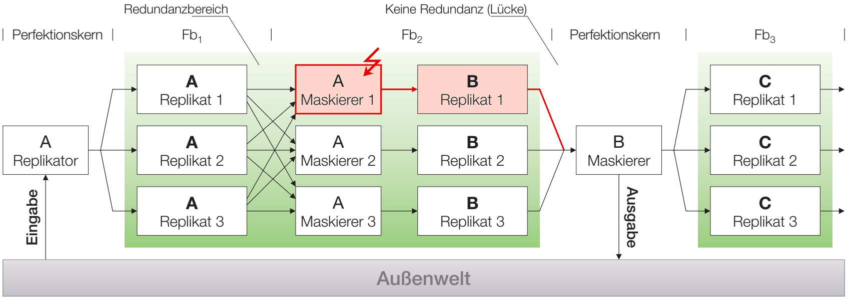
Aktuelle Forschung @ I4

Probleme der arithmetischen Codierung

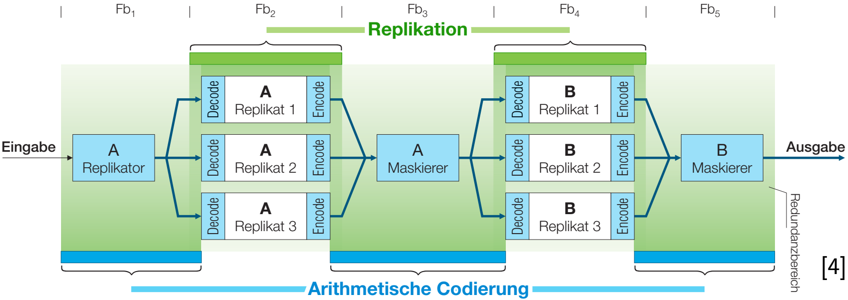
- die codierte Ausführung kompletter Programme ist derzeit zu teuer
- Fehlerfortpflanzung über Berechnungen möglich
- hohe Bandbreite für die Fehlerdiagnose (fehlerfreie Prüfinstanz?)

Probleme der Replikation

- kritische Fehlerstellen in der Infrastruktur (s. Folie IV/**??**)
- Unvollständigkeit (Lücken) der Redundanz (Redundanzbereich)



Combined Redundancy – *CoRed* [5]



[4]

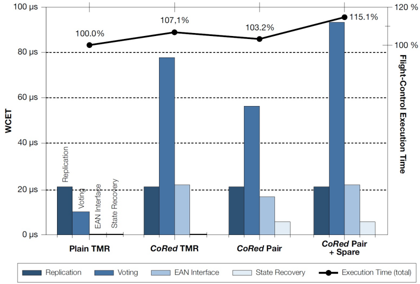
☞ selektiver Einsatz der Codierung erscheint hingegen vielversprechend

☞ genau diesen Weg beschreitet CoRed

- die eigentlich Berechnung wird durch Redundanz geschützt
- die kritischen Bruchstellen werden arithmetisch codiert
- 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

Combined Redundancy (CoRed) (Forts.)

Quelle Grafik: [5]



CoRed

selektive Anwendung von arithmetischer Codierung

- ⇝ 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 Codierung 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%

⇝ selektive Anwendung arithmetischer Codierung bringt Kostenvorteile

Gliederung

Zusammenfassung

Fehlererkennung möglichst ohne redundante Ausführung

- Erkennung von Operanden-, Berechnungs- und Operatorfehlern

$\leadsto$ Einsatz räumlicher Redundanz durch Prüfbits

arithmetisch Codierung

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

AN-Codierung $\leadsto$ Fehler im Wertbereich

- Codierung: Multiplikation mit einem konstanten Faktor A
- codierte Addition, Subtraktion, Multiplikation, Division
- Aussagenlogik, Schiebeoperatoren, Fließkommaarithmetik

ANBD-Codierung erweitert die AN-Codierung

- um statische Signaturen und dynamische Zeitstempel
- Codierung des Kontrollflusses $\leadsto$ Signaturen für Grundblöcke

CoRed-Ansatz $\leadsto$ selektive Anwendung der ANBD-Codierung

- durchgehende arithmetische Codierung wäre zu teuer

Literaturverzeichnis

[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 ANBDmem-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

Literaturverzeichnis (Forts.)

[4] Ulbrich, P. :
Ganzheitliche Fehlertoleranz in eingebetteten Softwaresystemen,
Friedrich-Alexander-Universität Erlangen-Nürnberg, Diss., 2014

[5] 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).*
Washington, DC, USA : IEEE Computer Society Press, Mai 2012. –
ISBN 978–1–4673–0938–7, S. 49–60

[6] 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