Verlässliche Echtzeitsysteme

Codierung

Peter Ulbrich

Lehrstuhl für Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität Erlangen-Nürnberg

`https://www4.cs.fau.de`

13. Mai 2019

Überblick

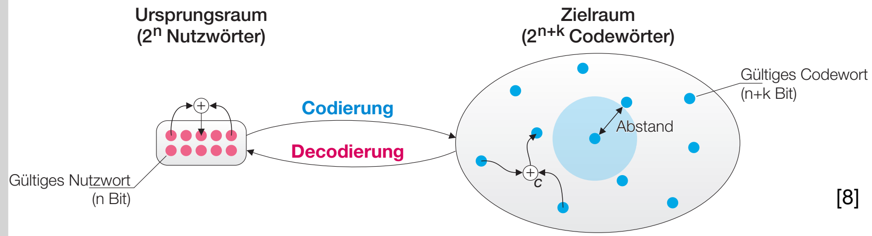
- Letztes Kapitel: Replikation ⇝ Grobe Granularität
  → Zum Zweck[1] der Fehlermaskierung (vgl. IV/12)
  - Im Allgemeinen durch Mehrheitsentscheid ($f + 2$ Replikate)
  - Durch einfache Replikation ($f + 1$ Replikate) im Falle von fail-silent-Verhalten
  - Hardwarebasiert durch redundanter Rechenknoten (vgl. IV/23 ff)
  - Softwarebasiert mittels Prozessen (vgl. IV/31 ff)
- ☞ Heute: Codierung ⇝ Feine Granularität
  → Zum Zweck[1] der Fehlererkennung
  - Implementierung von fail-silent-Verhalten
  - Systematische Nutzung von Informationsredundanz
  - Auf der Ebene einzelner Instruktionen und Datenelemente
  - Arithmetische Codierung von Werten und Berechnungen
- ☞ Maßgeschneiderte Anwendung von Redundanz ⇝ kombinierter Einsatz
  - Ergänzung der Stärken, Eliminierung der Schwächen

[1] Im Kontext dieser Veranstaltung, ohne Beschränkung der Allgemeinheit.

Gliederung

Allgemeines Grundprinzip der Codierung

Koordinierter Einsatz von Informationsredundanz

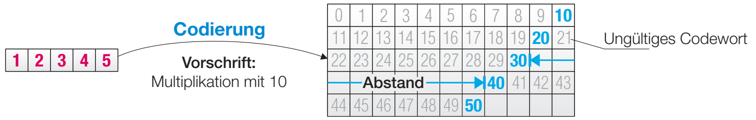


[8]

- Ausgangspunkt: Darstellung der Nutzdaten mithilfe von n Bits
- ☞ Codierung (engl. *error coding*) der Nutzdaten
  - Hinzufügen von k Prüfbits ↦ Einbringen von Informationsredundanz
  - Weiterhin 2^n gültige Codeworte bei nunmehr 2^{n+k} möglichen Worten
  → Fehler verlieren sich im ungenutzten Teil des Zielraums
- ⚠ Es genügt eine Instanz für die Fehlererkennung ≠ Replikation

Die Codierungsvorschrift

Transformation und Akzeptanztest: Ein einfaches Beispiel



[8]

- Codierungsvorschrift (engl. *encoding scheme*)
  - Vorschrift zur Überführung Ursprungsraum ⟷ Zielraum
  - ⚠ Variantenvielfalt (Paritätsbits [3], CRC [4], ...) ↦ anwendungsspezifisch
- Fehlererkennung mittels Akzeptanztest (vgl. IV/9)
  - Im Sinne eines Soll-Ist-Vergleichs ⇝ Konformität mit Codierungsvorschrift
  - Testbedingungen sind hierbei sowohl notwendig als auch hinreichend
  - → Zuverlässige Fehlererkennung
- Beispiel: Multipliziere mit 10
  - Codierung durch Multiplikation, Decodierung durch Division
  - Testbedingung: Division <u>ohne</u> Rest

Die Restfehlerwahrscheinlichkeit

Fehler bleiben unentdeckt, wenn ...

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

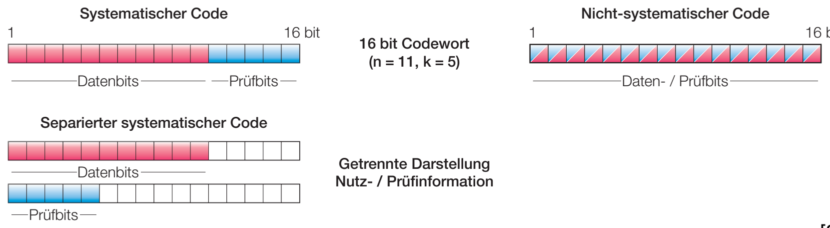
$$p_{sdc} = \frac{\text{Anzahl gültiger Nachrichten}}{\text{Anzahl möglicher Worte}} \approx \frac{2^n}{2^{n+k}} = 2^{-k}$$

  - Sofern man eine Gleichverteilung der Fehler zugrunde legt
  - → Stärke der Absicherung hängt direkt an der Zahl k redundanter Bits
- Bezogen auf die 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 (Codewörtern) sind also x fehlerhaft
    - → Diese werden durch die Codierung nicht erkannt

Codierung: Darstellung der Codewörter



[8]

- ⚠ Für die Integration der Prüfbits gibt es verschiedene Möglichkeiten [5]
- 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 (stets systematisch)
  - Getrennte Berechnung des funktionalen Anteils und der Prüfbits
  - Nicht-separierte Codes berechnen beides mit derselben Operation
- Systematische, nicht-separierte Codierung ist attraktiv
  - Behandlung des funktionalen Anteils/der Prüfbits in derselben Operation
  - Keine Decodierung beim Zugriff auf den funktionalen Anteil

Fehlermodell: Was kann alles schief gehen?

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

  return result;
}
```

- Was kann hier alles schief gehen?
- Transiente Fehler können folgende Fehler [2] hervorrufen:
  1. Operandenfehler (`a`, `b`, `c`, `result`)
     - Der Wert des Operanden wird verfälscht oder ist veraltet
     - Der Operand selbst wird verfälscht ⇝ falsche(s) Speicherstelle/Register
  2. Berechnungsfehler (`4 + 5 = 7`)
     - Die Operation erzeugt ein falsches Ergebnis
  3. Operatorfehler (`result = a` $\cancel{+} \rightarrow *$ `b`)
     - Der Programmzähler/die Instruktion wird verfälscht
     - → Ausführung einer falschen Instruktion
- ⚠ Datencodierung alleine bietet keine ausreichende Fehlererfassung

Gliederung

Arithmetische Codierung: AN-Codes

☞ Arithmetische Codierung: Erkennung von Berechnungsfehlern

- 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
    - → Absicherung gegen Fehler im Wertbereich

- Decodierung 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

⚠ AN-Codierung ist nicht-systematisch und nicht-separiert

AN-Codes: Operatoren

⚠ Die Codierung eines Programms erfolgt vor dessen Laufzeit
- Codierungsschlüssel A ist zur Laufzeit fest
- 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 ein codierter Operator $\circ_c$ nötig
- Dieser muss sowohl die Prüfbits als auch den funktionalen Teil v umfassen

- 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[2]	$z_c = \lfloor x_c /_c y_c \rfloor$	$Az = \lfloor (A \cdot Ax)/Ay \rfloor$	$A\lfloor x/y \rfloor$

[2] Umsetzung ist knifflig! Siehe [6, S.64ff].

AN-codierte 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 $\leadsto$ $(Ax \cdot y)$ 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
- → 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	$z_c = !_c\, x_c$	$z_c = 1_c -_c x_c$	$A(1 - x)$

→ Diese einfachen Operationen erfordern teils 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
- → Auch hier werden teure Berechnungsverfahren nötig
  - Diese greifen auf die codierten Standardoperatoren zu

⚠ Restfehlerwahrscheinlichkeit: Wähle ein geeignetes A!

Die Binärdarstellung stellt besondere Anforderungen.

⚠ Bitkipper können gültige Codewörter erzeugen $\leadsto p_{sdc}$

- Die Wahrscheinlichkeit hängt vom Abstand der Codewörter hab
- Ist jedoch nie Null

Der Codierungsschlüssel A bestimmt die Robustheit

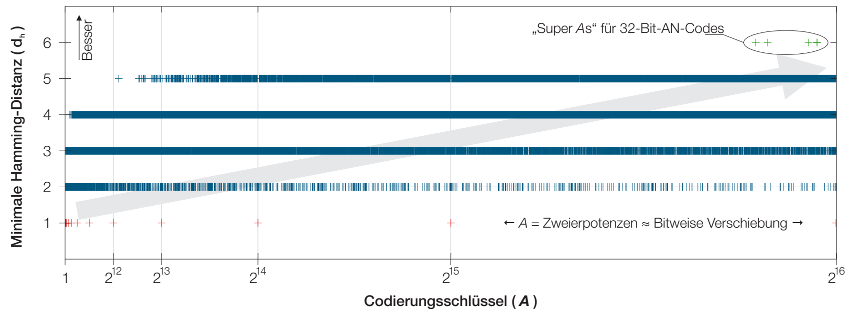
- Aus mathematischer Sicht sinnvoll: große Primzahlen
- Codierte Datenströme sollen möglichst teilerfremd sein

In der Praxis entscheidend: robuste Bitmuster

- Für binär-codierte Daten hängt dies von der Hammingdistanz d_h ab
- Erfreuliche Eigenschaft: $d_h - 1$ Bitfehler werden sicher erkannt
- → An wievielen Bitpositionen unterscheiden sich zwei Nachrichten

Wähle ein geeignetes A! (Forts.)

Experimentelle Bestimmung der Hamming-Distanz



[8]

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

Grenzen der AN-Codierung

- AN-Codes decken Fehler im Wertebereich vollständig ab

⚠ Fehlererfassung ist jedoch immer noch unvollständig

- 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
  - → 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
  - → 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?
- → ANB-Codes

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 Decodierung

$$v_c \mod A = B_v \quad v = (v_c - B_v)/A$$

- 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 Berechnungsergebnis 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

```
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_c;
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: $B_{result} \neq B_x + B_b + B_c$
    - → Eine Erkennung des Fehlers ist gewährleistet
  - Subtraktion statt einer Addition in Zeile 3
    - Die Signatur würde sich ändern: $B_{result} \neq B_a + B_b - B_c$
    - → Eine Erkennung des Fehlers ist gewährleistet

⚠ Keine Fehlererfassung auf der zeitlichen Achse

⚠ The Vital Coded Processor (VCP, [2])

Bislang vollständigste Variante der arithmetschen Codierung

☞ Forin erweitert den Ansatz um Zeitstempel $D \rightsquigarrow$ ANBD-Codes

- Ursprünglich: ein durch ANBD-Codierung geschützter Prozessor
  - Teilweise werden Elemente direkt in Hardware implementiert
    - En- bzw. Decodieren 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 (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 entsprechend mit veralteten Daten statt

☞ Lösung: „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

☞ Vollständige Abdeckung aller auf Folie 8 angenommen Fehler
  - 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

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

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

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

⚠ Gemeinsamer Operanden (hier: y_c) und Berechnungen in beiden Zweigen (Grundblöcken) notwendig

Indirekte Codierung des Kontrollflusses [7]

☞ Idee: Jeder Grundblock *x* bekommt eine explizite Signatur BB_x
- BB_x umfasst die Summe aller (regulären) Signaturen im Grundblock *x*

- Überprüfung zur Laufzeit durch Funktionswächter (engl. *Watchdog*)
  - Die Anwendung sendet die ermittelte Signatur BB_x an den Wächter
  - Er besitzt ein Feld `s` der zu erwartenden Werte BB_i
- Dynamische Berechnung von BB_x mittels Zählvariable `acc`
  - Wert am Beginn des Grundblocks: `acc` $=$ `s[i]` $- BB_x - x_{id}$
    - `s[x]` enthält den erwarteten Wert nach dem Grundblock *x*
    - Die statisch bestimmte Signatur BB_x wird abgezogen
    - Ebenso eine eindeutige ID x_{id} ⇝ bedingte Sprünge
  - `acc` wird kontinuierlich um die jeweils bestimmte Signatur inkrementiert
  - Für den nachfolgenden Grundblock wird `acc` neu initialisiert

⚠ Ansatz erfordert keine gemeinsamen Operanden jedoch einen vertrauenswürdigen (Hardware-) Funktionswächter für `s`

Codierung sequentieller Instruktionsfolgen

Wie sieht das aus? Erläuterung anhand eines Beispiels aus [7].

- Uncodierter Grundblock:

```
1  bb1:
2    x = a + b
3    y = x - d
4    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. Überwachung von bb_1 vorbereiten
   - Zu Beginn gilt: acc $=$ s$[$i$]$ $- BB_{bb1} - bb1_{id}$
   - $BB_{bb1} = (B_a + B_b) + (B_a + B_b - B_d)$
2. Codierung der Berechnungen (Zeile 2 und 4)
3. Aufbau der Signatur BB_{bb1} in `acc`
   - Zeile 3: acc $=$ s$[$i$]$ $- BB_{bb1} - bb1_{id} + B_a + B_b$
   - Zeile 5: acc $=$ s$[$i$]$ $- bb1_{id}$ (vereinfacht $+x_c - d_c$)
4. Signatur an den „Watchdog“ senden (Zeile 7)
5. Vorbereitungen für den Grundblock bb2
   - Zeile 8: `acc` $=$ `s[i + 1]` $- bb1_{id}$
   - Zeile 12: `acc` $=$ `s[i]` $- BB_{bb2} - bb2_{id}$

Codierung bedingter Sprünge [7]

- Herausforderung: Übertragung des Konzepts für bedingten Code
  → Wie funktioniert hier die Umschaltung zwischen Grundblöcken?
  - Welcher der nächste Grundblock ist, hängt ja vom bedingten Sprung ab …
- ⚠ Wiederrum gilt: Ungeschützte Stellen
  - Das Ergebnis der Entscheidung könnte verfälscht werden
  - Der bedingte Sprung selbst könnte verfälscht werden
- ☞ Das Ergebnis der Entscheidung muss absichert werden
  → Hier hilft es, dieses Ergebnis arithmetisch zu codieren
- ☞ Korrektheit des bedingten Sprungs muss sichergestellt werden
  → Hier helfen die IDs der angesprungenen Grundblöcke
  - Sind vorab bekannt ⇝ 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 [7].

```
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: $\texttt{acc} = \texttt{s[i]} - BB_{bb1} - bb1_{id}$
2. Zeile 2-3: Bedingung wird codiert ⇝ $cond_c$
   - wahr ⇝ $cond_c = A \cdot 1 + B_{cond}$ und falsch ⇝ $A \cdot 0 + B_{cond}$
3. Zeile 4: sende $\texttt{acc} = \texttt{s[i]} - BB_{bb1} - bb1_{id} + B_{cond}$ an den „Watchdog"
4. Zeile 6-8: bereite `acc` für den Sprung auf `bbt` vor
   → Nun gilt $\texttt{acc} = \texttt{s[i]} - BB_{bbt} - bbt_{id} - (A \cdot 1 + B_{cond})$
5. Zeile 10-12: extrahiere Wert von `cond_c` ⇝ aktualisiere `acc` und springe
   → Nun gilt $\texttt{acc} = \texttt{s[i]} - BB_{bbt} - bbt_{id} - (A \cdot 1 + B_{cond}) + cond_c$

Codierung bedingter Sprünge (Forts.)

Wie sieht das aus? Erläuterung anhand eines Beispiels aus [7].

```
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 $\texttt{cond} = \texttt{wahr}$ bleibt nichts zu tun, schließlich gilt $cond_c = A \cdot 1 + B_{cond}$
   → Insgesamt gilt: $\texttt{acc} = \texttt{s[i]} - BB_{bbt} - bbt_{id}$, der Anfangswert für den Grundblock `bbt`
7. für einen Sprung zu `bbf` ist jedoch eine Korrektur notwendig
   - Schließlich wurde `acc` für einen Sprung zu `bbt` vorbereitet
8. Zeile 4: eingangs gilt $\texttt{acc} = \texttt{s[i]} - BB_{bbt} - bbt_{id} - A \cdot 1$
   - Hier gilt $cond_c = A \cdot 0 + B_{cond} = B_{cond}$
   → Korrigiert: $\texttt{acc} = \texttt{s[i]} - BB_{bbf} - bbf_{id}$, der Anfangswert für des Grundblocks `bbf`
9. nun kann weiter zu `bbf` gesprungen werden

Software Encoded Processor (SEP, [10])

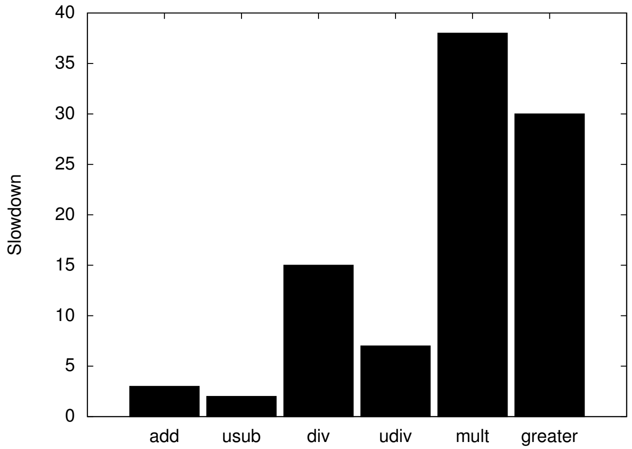
Programme codiert in einem Interpreter ausführen

- 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 ⇝ Fehlererkennungsrate ist sehr gut
  - Codierter Interpreter: keine fehlerhaften Ergebnisse
  - Nicht-codierte Ausführung:
    - Interpretiert: 4% der Ergebnisse fehlerhaft
    - Native Ausführung: 9% der Ergebnisse fehlerhaft
    → 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, [7])

Quelle Grafik: [7]

- Codierung wird vor der Laufzeit durch einen Compiler durchgeführt
  - Nicht mehr zur Laufzeit durch einen Interpreter

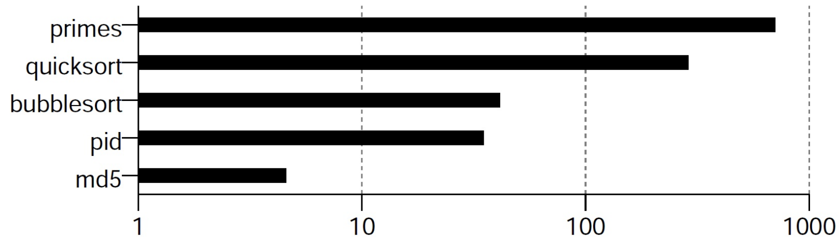
☞ Hierfür muss aber der Quelltext vorhanden sein

- 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



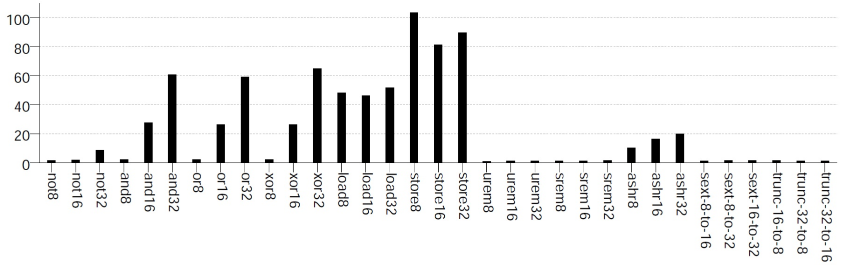
Compiler Based Encoding (CBE) (Forts.)

Quelle Grafik: [1]

⚠ Vergleich mit nativ ausgeführten Operationen

→ Bringt die wahren Laufzeitkosten zutage

- Operationen, die nicht direkt codierbar sind:

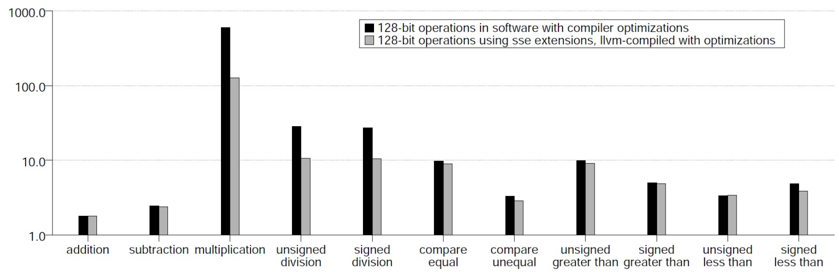


- Das Speichern eines 8 Bit großen Wortes ist bis zu 100x langsamer
  - Diese Operation besteht aus diversen Einzelschritten
  - Laden, bitweises Und, Schiebeoperation, ...
  - → All das muss in codierter Form ablaufen, all das ist teuer

Compiler Based Encoding (CBE) (Forts.)

Quelle Grafik: [1]

- Direkt codierbare arithmetische Operationen



- Auch hier sind Laufzeitkosten zum Teil beträchtlich
  - Addition und Subtraktion sind vergleichsweise günstig
  - Einfache Vergleichsoperationen sind aber relativ teuer
  - Multiplikation und Division benötigen teure 128-Bit-Operationen

Gliederung

Schwächen der Codierung

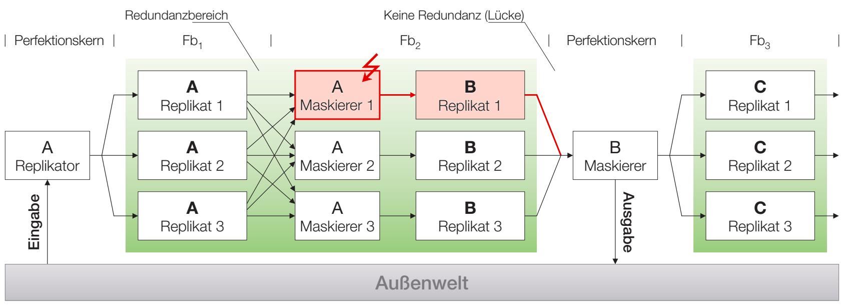
⚠ Arithmetischen Codierung ist (derzeit) unzureichend für die Härtung kompletter Programme

- Sehr hoher Ressourcenbedarf
  - Beispiel Laufzeitkosten von 500 – 1000 % vs. ≈300 % für Replikation
  - Hohe Bandbreite[3] für die Fehlerdiagnose
- Fehlerfortpflanzung schwer zu unterbinden
  - Kontrollflussüberwachung in der Praxis unzuverlässig
  - Alternative: fehlerfreie Prüfinstanz (Perfektionskern)
  - Schutzwirkung bei fortgesetzter Verwendung fehlerhafter Werte unsicher
  - → Zusätzliche Bandbreite für Fehlerdiagnose ⇝ Kosten
- Restfehlerwahrscheinlichkeit (vgl. Folie 6)
  - Schwere des Fehlers hat Einfluss auf die Erkennungsleistung
  - ⚠ Konzeptbedingte Eigenschaft der Codierung

[3] Für die Testverfahren notwendige Ressourcen (Rechenzeit, Kommunikation, . . .) im Vergleich zur eigentlichen Systemfunktion.

Schwächen der Replikation

Kritische Fehlerstellen in der strukturellen Redundanz

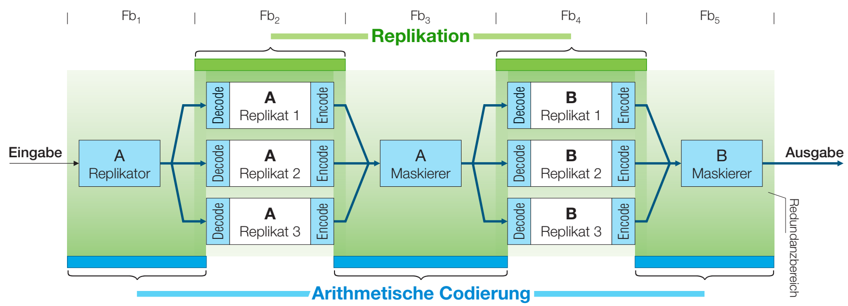


⚠ Vollständige Replikation ist typischerweise unmöglich
- Ergebnisse müssen (irgendwann) konsolidiert werden
- Insbesondere problematisch bei softwarebasierter Replikation
- → Unvollständigkeit der Redundanz (Lücken im Redundanzbereich)

- Kritische Fehlerstellen in der Infrastruktur (vgl. IV/21)
  - Zeitliche und räumliche Isolation (Betriebssystem?)
  - Eingangsreplikation & Mehrheitsentscheider

Combined Redundancy – *CoRed* [9]

Forschung @ I4



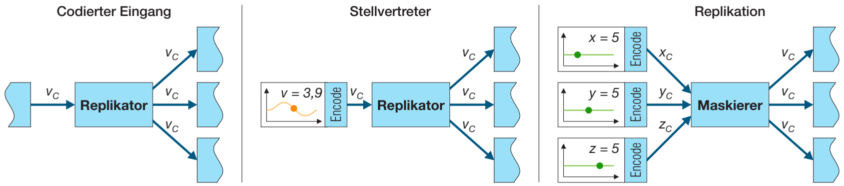
[8]

☞ Maßgeschneiderter Einsatz verschiedener Redundanzarten
- Stärken kennen und Schwächen verdecken

- Genau diesen Weg beschreiten wir mit Combined Redundancy (CoRed)
  - Die eigentliche Berechnung wird durch Redundanz geschützt
  - Die Replikationsinfrastruktur wird arithmetisch codiert

Technische Realisierung

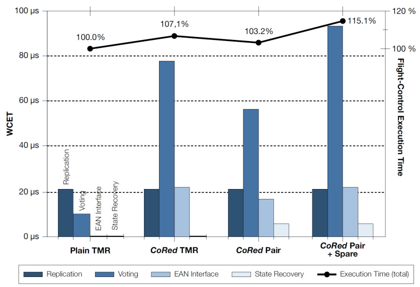
Am Beispiel der Eingangsreplikation



- Mehrheitsentscheider als codierter Operator ⇝ CoRed-Voter
  - Akzeptiert codierte Varianten (x_c, y_c, z_c) und liefert codierten Gewinner (v_c)
  - → Genaue Funktionsweise wird in der Übung besprochen
- Codierte Eingangsreplikation
  - Vorgehen wie gehabt (vgl. IV/19)
  - Schnellstmögliche Codierung der Eingaben
  - Akzeptanzmaskierer nutzt codierten Mehrheitsentscheider
- Idealerweise codierende Sensoren und Aktoren
  - Zu keinem Zeitpunkt ungeschützte Werte
  - → Lückenlose Fehlerdiagnose

Kostenbetrachtung

Quelle Grafik: [9]



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 codieren, wäre man hier bei mehreren 100%

Gliederung

Zusammenfassung

Fehlererkennung Möglichst ohne redundante Ausführung
- Erkennung von Operanden-, Berechnungs- und Operatorfehlern
- → Einsatz räumlicher Redundanz durch Prüfbits

Arithmetisch Codierung
- nicht-systematisch und nicht-separiert

AN-Codierung ⇝ Fehler im Wertebereich
- 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 ⇝ Signaturen für Grundblöcke

CoRed-Ansatz ⇝ 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.) ; Seyfarth, 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: *Proceedings of the IFAC IFIP/IFORS Symposium on Control, Computers, Communications in Transportation (CCCT '89)*, 1989, S. 79–84

[3] Hamming, R. W.:
Error detecting and error correcting codes.
In: *Bell System technical journal* 29 (1950), Nr. 2, S. 147–160

[4] Peterson, W. W. ; Weldon, E. J.:
Error-correcting codes.
2.
Cambridge, MA, USA : MIT Press, 1972. –
572 S. –
ISBN 978–0–2621–6039–1

Literaturverzeichnis (Forts.)

[5] Rao, T. R. N.:
Error Coding for Arithmetic Processors.
1.
Orlando, FL, USA : Academic Press, 1974. –
218 S. –
ISBN 978–0–1258–0750–0

[6] Schiffel, U. :
Hardware Error Detection Using AN-Codes, Technische Universität Dresden, Fakultät Informatik, Diss., 2011

[7] 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

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

Literaturverzeichnis (Forts.)

[9] 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

[10] 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