ulm university universität ulm

Compiler für Eingebettete Systeme

[CS7506]

Sommersemester 2014

Heiko Falk

Institut für Eingebettete Systeme/Echtzeitsysteme
Ingenieurwissenschaften und Informatik
Universität Ulm

ulm university universität ulm

Kapitel 4

Prepass-Optimierungen

Inhalte der Vorlesung

1. Einordnung & Motivation der Vorlesung
2. Compiler für Eingebettete Systeme – Anforderungen & Abhängigkeiten
3. Interner Aufbau von Compilern
4. **Prepass-Optimierungen**
5. HIR Optimierungen und Transformationen
6. Instruktionsauswahl
7. LIR Optimierungen und Transformationen
8. Register-Allokation
9. Compiler zur $WCET_{EST}$-Minimierung
10. Ausblick

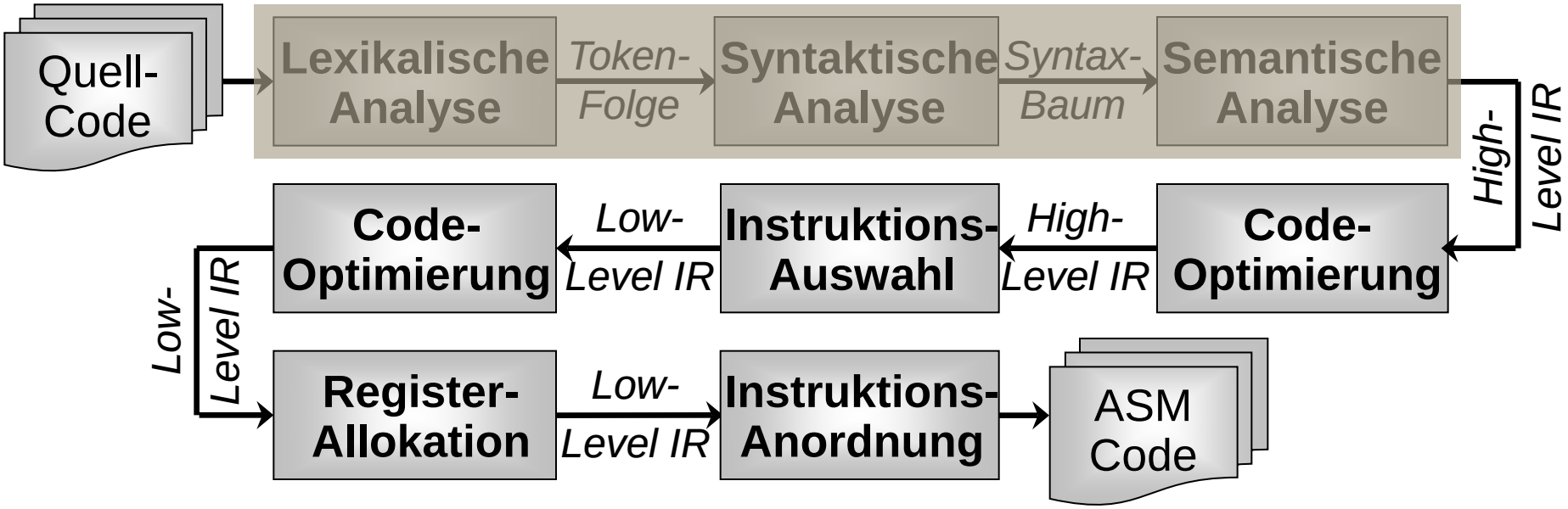
Inhalte des Kapitels

4. Prepass-Optimierungen

- Motivation von Prepass-Optimierungen
- *Loop Nest Splitting*
  - *Embedded Multimedia:* MPEG 4 *Motion Estimation*
  - Ablauf von *Loop Nest Splitting*
  - *Condition Satisfiability*
  - *Condition Optimization* & Genetische Algorithmen
  - *Search Space Generation*
  - *Search Space Exploration*
  - Ergebnisse (ACET, Energie, Codegröße)

Motivation von Prepass-Optimierungen

☞ ***Rückblick: Struktur eines Optimierenden Compilers mit 2 IRs:***



Frage: Darf nur der Compiler Code optimieren?

Motivation von Prepass-Optimierungen

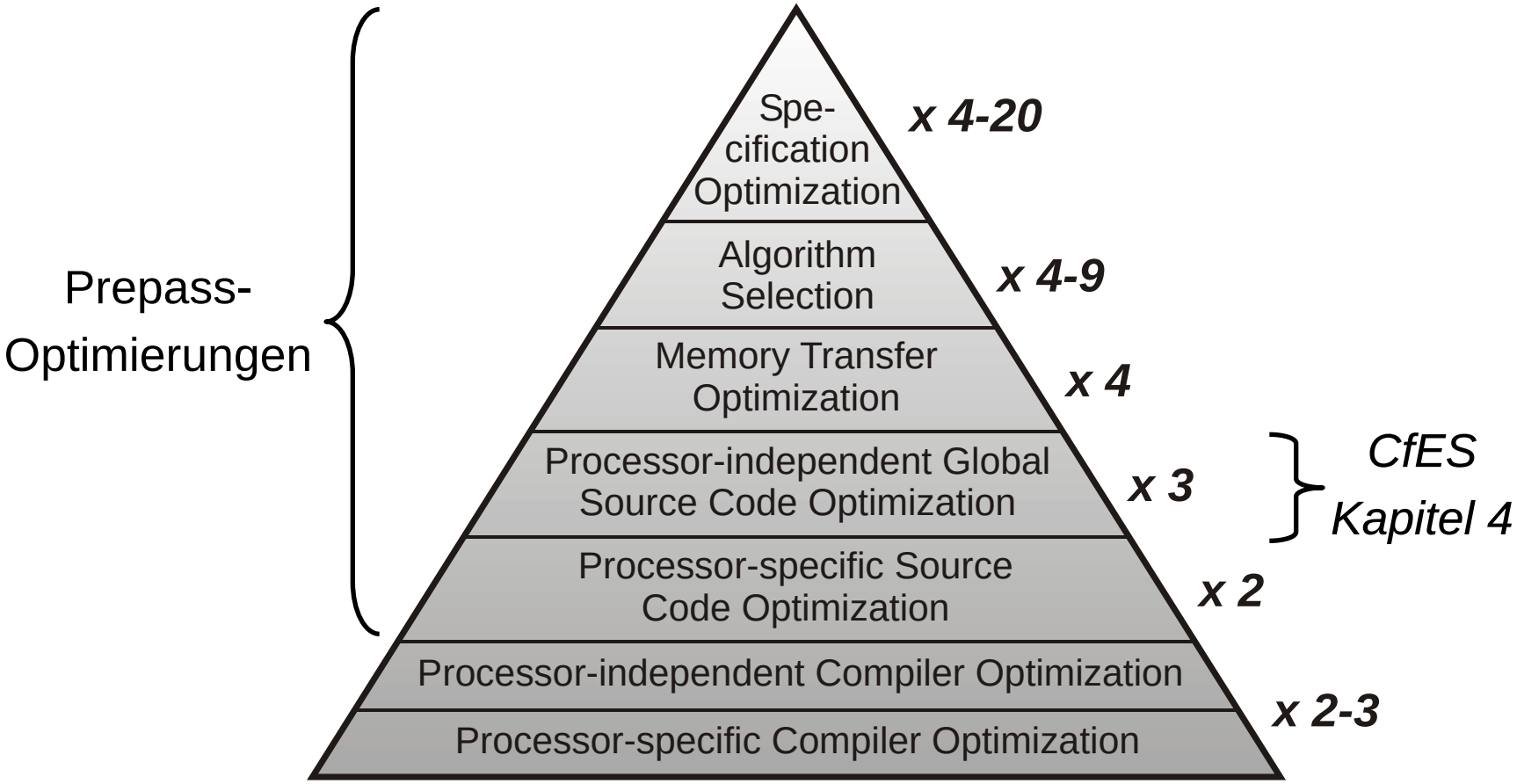
Optimierungen außerhalb des Compilers heißen

– *Postpass-Optimierung*, wenn sie *nach* dem Compiler ablaufen
– *Prepass-Optimierung*, wenn sie *vor* dem Compiler ablaufen

Vorteile von Prepass-Optimierungen

– Quellcode-Transformation leichter verständlich & zugänglich
– Erlaubt manuelles „Ausprobieren“ einer Optimierungstechnik vor einer aufwändigen Implementierung.
– Wegen Quellcode-Niveau unabhängig vom aktuellen Compiler; prinzipiell für jeden Compiler der Quellsprache anwendbar.
– Wegen Quellcode-Niveau unabhängig von einer festen Zielarchitektur; prinzipiell für beliebige Architekturen anwendbar.

Abstraktionsebenen von Optimierungen



Inhalte des Kapitels

4. Prepass-Optimierungen

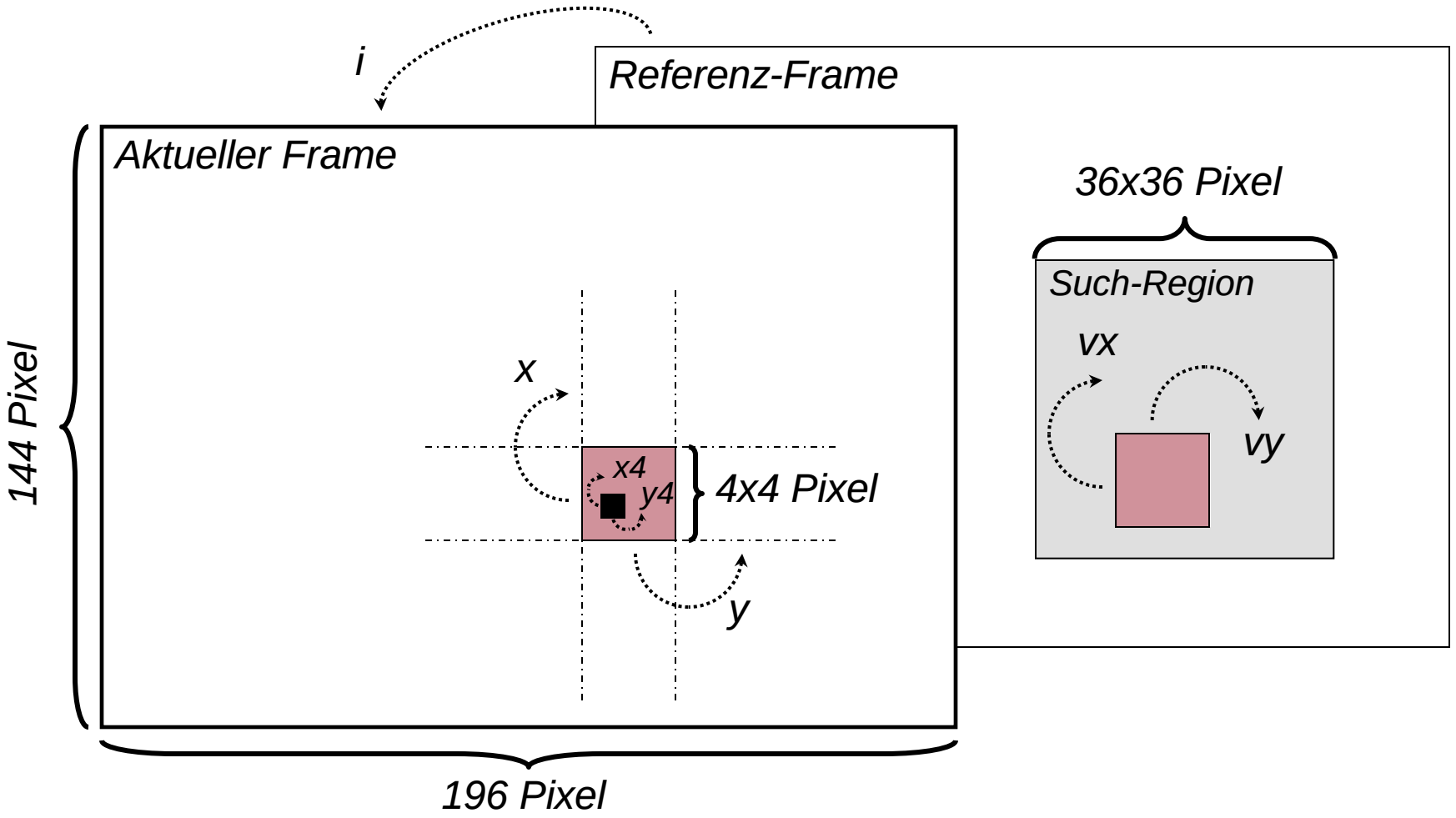
- Motivation von Prepass-Optimierungen
- *Loop Nest Splitting*
  - *Embedded Multimedia:* MPEG 4 *Motion Estimation*
  - Ablauf von *Loop Nest Splitting*
  - *Condition Satisfiability*
  - *Condition Optimization* & Genetische Algorithmen
  - *Search Space Generation*
  - *Search Space Exploration*
  - Ergebnisse (ACET, Energie, Codegröße)

Einsatzgebiet von *Loop Nest Splitting*

Eingebettete Multimedia-Anwendungen

- *Datenfluss-dominiert*, d.h. erhalten große Datenmengen als Eingabe, berechnen große Datenmengen als Ausgabe
  (im Gegensatz zu kontrollfluss-dominierten Steuerungs- und Regelungsanwendungen).
- Hauptteil der Laufzeit in (tief) *verschachtelten Schleifen* verbracht
- *Einfache Schleifenstrukturen* mit bekannten bzw. durch Compiler analysierbaren Unter- und Obergrenzen
- Manipulation großer mehrdimensionaler Felder
- Typisches Beispiel: *Streaming*-Anwendungen wie MPEG4

Beispiel: MPEG4 *Motion Estimation*



Quellcode MPEG4 *Motion Estimation*

```
for (i = 0; i < 20; i++)
  for (x = 0; x < 36; x++)
    for (y = 0; y < 49; y++)
      for (vx = 0; vx < 9; vx++)
        for (vy = 0; vy < 9; vy++)
          for (x4 = 0; x4 < 4; x4++)
            for (y4 = 0; y4 < 4; y4++) {
              if (4*x+x4<0 || 4*x+x4>35 ||
                  4*y+y4<0 || 4*y+y4>48)
                then_block_1; else else_block_1;
              if (4*x+vx+x4-4<0 || 4*x+vx+x4-4>35 ||
                  4*y+vy+y4-4<0 || 4*y+vy+y4-4>48)
                then_block_2; else else_block_2; }
```

Beobachtungen

Bei Übersetzung und Ausführung dieses Quellcodes

- Insgesamt werden 91.445.760 *If-Statements* ausgeführt.
- Wegen *If-Statements* sehr unregelmäßiger Kontrollfluss.
- Zusätzlicher arithmetischer Aufwand:
  Multiplikationen, Additionen, Vergleiche, logisches Oder, ...

☞ *Performance dieses Codes durch Kontrollfluss eingeschränkt, nicht durch Berechnung der Bewegungsvektoren!*

Loop Nest Splitting

Automatisierte Schleifen- & *If-Statement*-Analyse

– **x**, **y**, **x4** und **y4** nehmen niemals Werte ein, so dass Bedingungen **`4*x+x4<0`** und **`4*y+y4<0`** jemals wahr wären.

☞ *Bedingungen können durch konstanten Wahrheitswert* '**`0`**' *ersetzt werden.*

– Für **x** ≥ 10 oder **y** ≥ 14 sind beide *If-Statements* zwingend erfüllt, so dass deren *then*-Zweige ausgeführt werden.

☞ *Für mehr als 92% aller Ausführungen der innersten* **y4***-Schleifen sind beide If-Statements wahr.*

Quellcode nach *Loop Nest Splitting*

```
for (i = 0; i < 20; i++)
  for (x = 0; x < 36; x++)
    for (y = 0; y < 49; y++)
      if (x >= 10 || y >= 14)                    // Splitting-If
        for (; y < 49; y++)                  // Zweite y-Schleife
          for (vx = 0; vx < 9; vx++) ... {       // Keine
            then_block_1; then_block_2; }     // If-Stmts
      else
        for (vx = 0; vx < 9; vx++) ... {
          if (0 || 4*x+x4>35 || 0 || 4*y+y4>48)  // Alte
            then_block_1; else else_block_1; // If-Stmts
          if (4*x+vx+x4-4<0 || 4*x+vx+x4-4>35 || ...
            then_block_2; else else_block_2; }
```

Optimierte Code-Struktur

Splitting-If

– Wann immer die Bedingung des *Splitting-If* erfüllt ist, sind automatisch die Bedingungen aller ursprünglichen *If-Statements* erfüllt.

☞ *Then*-Zweig des *Splitting-If* enthält keine originalen *If-Statements* mehr, sondern nur noch deren *then*-Zweige.

– Wenn Bedingung des *Splitting-If* nicht erfüllt ist, ist keine Aussage über Wahrheitswert der ursprünglichen *If-Statements* möglich.

☞ *Else*-Zweig des *Splitting-If* enthält alle originalen *If-Statements*, um Korrektheit des Codes zu erhalten.

Warum Zweite Y-Schleife?

Intuitiver Code:

```
for (x=0; x<36; x++)
  for (y=0; y<49; y++)
    if (x>=10 || y>=14)
      for (vx=0; vx<9; vx++) ...
```

☞ ***Splitting-If:***
1 Ausführung für
jedes einzelne **y** ∈ [14, 48]

```
y = 16
```

Optimierter Code:

```
for (x=0; x<36; x++)
  for (y=0; y<49; y++)
    if (x>=10 || y>=14)
      for (; y<49; y++)
        for (vx=0; vx<9; vx++) ...
```

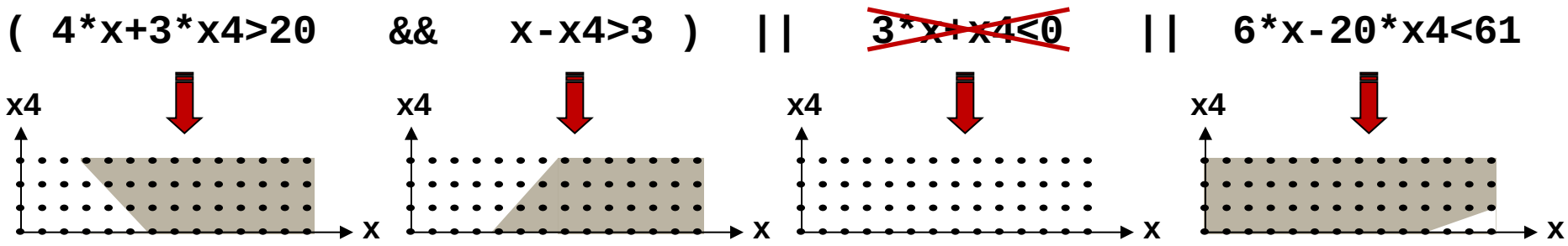
☞ ***Splitting-If:***
1 Ausführung für
alle **y** ∈ [14, 48] *zusammen*

```
y = 16
```

Phasen von *Loop Nest Splitting*

- ***Condition Satisfiability:*** Finden einzelner stets erfüllbarer oder unerfüllbarer Bedingungen in *If-Statements*
- ***Condition Optimization:*** Für jede einzelne Bedingung *C*, finde eine einfacher auszudrückende Bedingung *C'*, so dass *C'* $\Rightarrow$ *C* gilt (wenn *C'* wahr ist, ist auch *C* wahr).
- ***Search Space Generation:*** Verknüpfe alle einzelnen Bedingungen *C'* zu einer Struktur *G*, die alle *If-Statements* mitsamt deren Struktur (**&&**, **||**) modelliert.
- ***Search Space Exploration:*** Bestimme anhand von *G* eine Bedingung für das *Splitting-If*, die zur Minimierung der Ausführung von *If-Statements* führt.

Ablauf von *Loop Nest Splitting* (1)

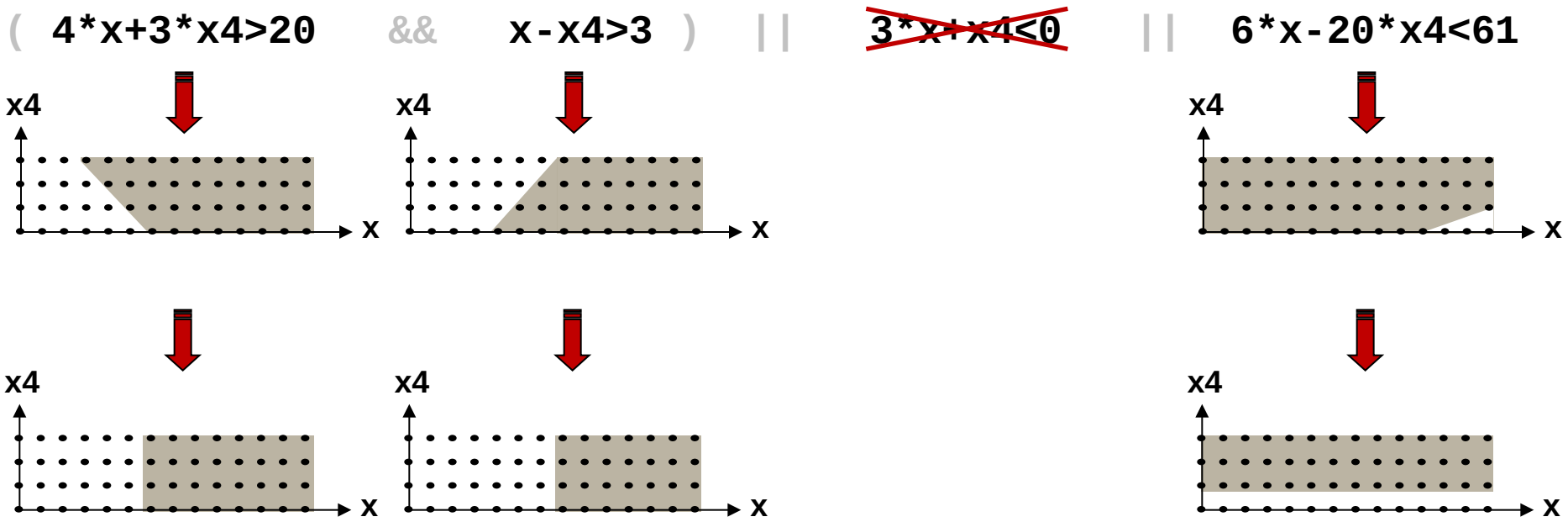


1 - *Condition Satisfiability*

☞ ***Hinweis:***
Dieses Beispiel entspricht nicht dem MPEG4-Code der vorigen Folien!

– Angenommene Schleifengrenzen:
$0 \leq \mathbf{x} \leq 13$
$0 \leq \mathbf{x4} \leq 3$

Ablauf von *Loop Nest Splitting* (2)

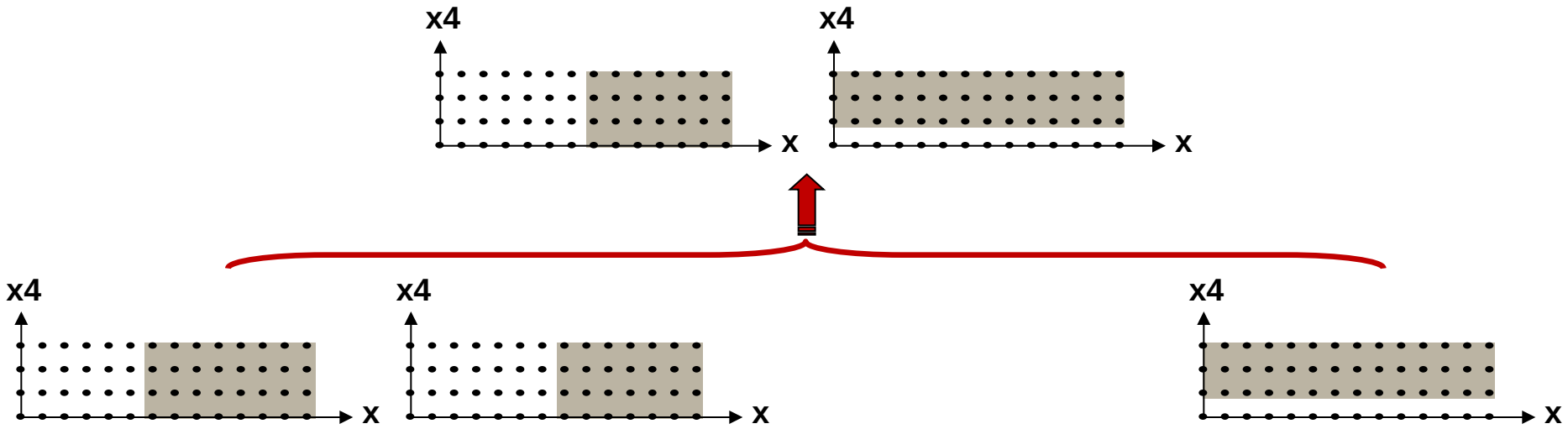


1 - *Condition Satisfiability*

2 - *Condition Optimization*

Ablauf von *Loop Nest Splitting* (3)

```
( 4*x+3*x4>20  &&  x-x4>3 )  ||  3*x+x4<0  ||  6*x-20*x4<61
```



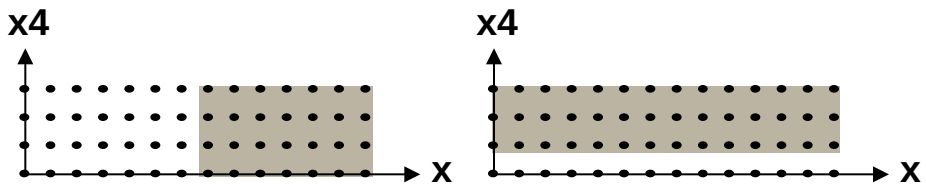
1 - *Condition Satisfiability*

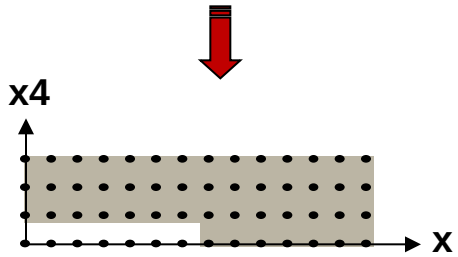
2 - *Condition Optimization*

3 - *Search Space Generation*

Ablauf von *Loop Nest Splitting* (4)

```
( 4*x+3*x4>20   &&   x-x4>3 )  ||  3*x+x4<0  ||  6*x-20*x4<61
```






```
x>=7  ||  x4>=1
```

1 - *Condition Satisfiability*

2 - *Condition Optimization*

3 - *Search Space Generation*

4 - *Search Space Exploration*

Voraussetzungen

Grenzen der Schleifen L

– Alle unteren und oberen Grenzen (l_L, u_L) sind konstant

If-Statements

– Sequenz schleifenabhängiger Bedingungen, mit logischem UND bzw. ODER verknüpft

  – Format: `if (` $C_1 \otimes C_2 \otimes \ldots$ `)` $\otimes \in \{\&\&, ||\}$

Schleifenabhängige Bedingungen

– Lineare Ausdrücke über Indexvariablen i_L der Schleifen

  – Format: $C_x \simeq \sum_{L=1}^{N} (c_L * i_L) + c \geq 0 \qquad c_L, c \in \mathbb{Z}$

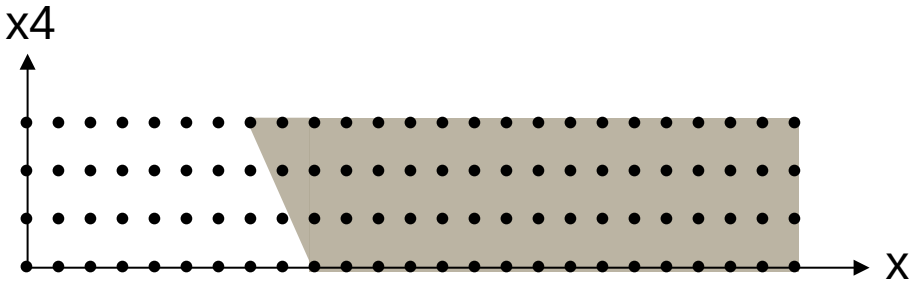
Polytope & Lineare Bedingungen

Definition (Polyeder / Polytope)

- *Polyeder* $P = \{x \in \mathbb{Z}^N | Ax \geq b\}$ $\quad A \in \mathbb{Z}^{m \times N}, b \in \mathbb{Z}^m$
- Polyeder P heißt *Polytop* genau dann wenn $|P| < \infty$

Beispiel: Modell linearer Bedingungen in verschachtelten Schleifen

- **4*x + 3*x4 > 35** für **x** $\in [0, 35]$, **x4** $\in [0, 3]$ als Polytop
- $$P = \{p \in \mathbb{Z}^2 | \begin{pmatrix} 4 & 3 \\ 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} p \geq \begin{pmatrix} 36 \\ 0 \\ -35 \\ 0 \\ -3 \end{pmatrix}$$



Konvexität von Polytopen

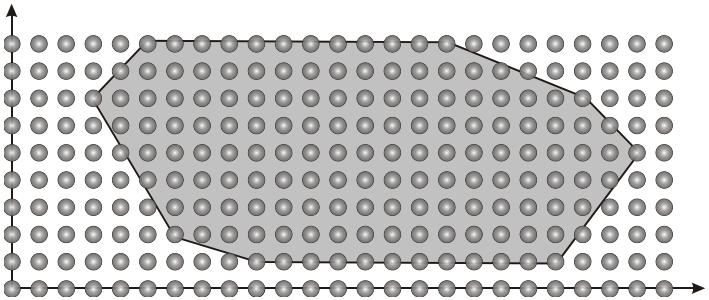
Definition (Konvexität)

– Eine Menge $S \subset \mathbb{Z}^N$ heißt *konvex*, wenn für alle $x, y \in S$ jede konvexe Kombination $ax + by$ von x und y $(a + b = 1; b \geq 0)$ in S ist.

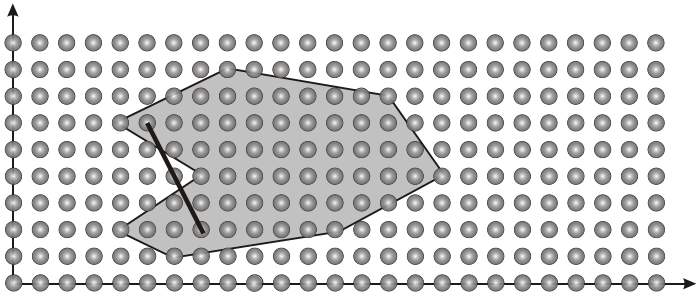
Informal

– Jede Linie zwischen zwei beliebigen Punkten x und y aus S muss vollständig in S liegen.

☞ *Polytope sind konvex.*



Konvex



Nicht konvex

Operationen auf Polytopen

Eigenschaften

– Die ***Schnittmenge*** zweier Polytope ist auch ein Polytop.

– Die ***Vereinigung*** zweier Polytope ist kein Polytop, da die resultierende Menge nicht konvex ist.

Definition (Endliche Vereinigung von Polyedern, EVP)

– Für Polyeder $P_1, \ldots, P_n (n \in \mathbb{N}, n < \infty)$ heißt $V := \{P_1 \cup \cdots \cup P_n\}$ *endliche Vereinigung von Polyedern (EVP)*.

Mengenoperationen auf EVPs $A = \bigcup_i A_i, B = \bigcup_j B_j$

– $A \cap B := (\bigcup_i A_i) \cap (\bigcup_j B_j) = \bigcup_{i,j} (A_i \cap B_j)$

– $A \cup B := (\bigcup_i A_i) \cup (\bigcup_j B_j)$

Phase 1 – *Condition Satisfiability*

Ziel

– Finden einzelner schleifenabhängiger Bedingungen C_x, die für alle Werte der Indexvariablen aller umgebender Schleifen konstant ‚*true*' oder konstant ‚*false*' ergeben.

Vorgehen

– Überführen jeder Bedingung C_x in ein Polytop P_x (☞ *Folie 23*)

– Test auf leere Menge: $P_x == \emptyset \Rightarrow C_x$ stets ‚*false*'

– Test auf Allmenge / Universum: $P_x == U \Rightarrow C_x$ stets ‚*true*'

Quellcode-Modifikation

– Ersetzung solcher konstant wahrer oder falscher Bedingungen im Quellcode durch Konstanten ‚**`0`**' bzw. ‚**`1`**'.

Phase 2 – *Condition Optimization*

Gegeben

– N Schleifen und eine Bedingung $C = \sum_{L=1}^{N} (c_L * i_L) + c \geq 0$

Angenommene hypothetische Situation

```
for ( loop L_1 )
  ...
    for ( loop L_N )
      if ( C ) ...
```

Annahme: Die Schleifen enthalten nur 1 If-Statement mit nur 1 Bedingung. Weitere If-Statements & Bedingungen sind quasi „ausgeblendet“.

Ziel: Bestimme Intervalle $[l_{C,1}, u_{C,1}] \ldots [l_{C,N}, u_{C,N}]$, so dass gilt:

– C ist erfüllt für alle Schleifen-Iterationen in diesen Intervallen

– Minimierung der Ausführung von *If-Statements* nach hypothetischem *Loop Nest Splitting* anhand dieser Intervalle.

Optimierung einer Bedingung *C*

$$C = \sum_{L=1}^{N} (c_L * i_L) + c \geq 0$$

- Bestimmung von Werten $l_{C,L}$ und $u_{C,L}$ für jede Schleife L
- Interpretation: C ist erfüllt für alle $l_{C,L} \leq i_L \leq u_{C,L}$
- Optimierungsziel:
  Alle Werte $l_{C,L}$ und $u_{C,L}$ minimieren Gesamt-Anzahl ausgeführter *If-Statements*

- Vereinfachung: Wegen Linearität von C folgt
  entweder $l_{C,L} = l_L$ oder $u_{C,L} = u_L$
- Konsequenz: Bestimmung nur eines Wertes $v_{C,L}$
  Entweder erfüllt $[v_{C,L}, u_{C,L}]$ oder $[l_{C,L}, v_{C,L}]$ Bedingung C

Ein paar Definitionen...

Gegeben: $C = \sum_{L=1}^{N} (c_L * i_L) + c \geq 0$ $\quad l_L, u_L \quad v_{C,L}$

– **Totaler Iterationsraum** *(#Ausführungen d. innersten Schleifenkörpers)*

$$TIR = \prod_{L=1}^{N} (u_L - l_L + 1)$$

– **Beschränkter Iterationsraum** *(#Ausführungen d. innersten Schleifenkörpers, eingeschränkt auf durch $v_{C,L}$ definierte Bereiche)*

$$BIR = \prod_{L=1}^{N} r_L$$

$$r_L = \begin{cases} u_L - l_L + 1 & \text{für } c_L = 0, \\ u_L - v_{C,L} + 1 & \text{für } c_L > 0, \\ v_{C,L} - l_L + 1 & \text{sonst} \end{cases}$$

– **Splitting-Schleife** *(Index der Schleife, wo Splitting stattfindet)*

$$\lambda = \min\{\ i \mid L_i \in \Lambda, r_L \neq u_L - l_L + 1\ \}$$

Veranschaulichung (1)

Schleife 1

```
for (i = 0; i < 20; i++)
  for (x = 0; x < 36; x++)
    for (y = 0; y < 49; y++)
      for (vx = 0; vx < 9; vx++)
        for (vy = 0; vy < 9; vy++)
          for (x4 = 0; x4 < 4; x4++)
            for (y4 = 0; y4 < 4; y4++) {
              if (4*x+vx+x4-4>35)
                then_block_1; else else_block_1; }
```

Schleife N=7

Beachte:

- Schleifen von 1 ... *N* von außen nach innen durchnummeriert.
- Nur 1 Bedingung im *if-Statement*, statt vieler wie auf Folie 11!

Veranschaulichung (2)

Schleife 1

```
for (i = 0; i < 20; i++)
  for (x = 0; x < 36; x++)
    for (y = 0; y < 49; y++)
      for (vx = 0; vx < 9; vx++)
        for (vy = 0; vy < 9; vy++)
          for (x4 = 0; x4 < 4; x4++)
*Schleife N=7*
            for (y4 = 0; y4 < 4; y4++) {
              if (4*x+vx+x4-4>35)
                then_block_1; else else_block_1; }
```

***TIR*:**

- Anzahl der Ausführungen des Codes in geschweiften Klammern.
- Hier: 20 * 36 * 49 * 9 * 9 * 4 * 4 = 45.722.880

Veranschaulichung (3)

Schleife 1

```
for (i = 0; i < 20; i++)
  for (x = 0; x < 36; x++)
    for (y = 0; y < 49; y++)
      for (vx = 0; vx < 9; vx++)
        for (vy = 0; vy < 9; vy++)
          for (x4 = 0; x4 < 4; x4++)
            for (y4 = 0; y4 < 4; y4++) {
              if (4*x+vx+x4-4>35)
                then_block_1; else else_block_1; }
```

Schleife N=7

Seien $v_{C,1} = 0; v_{C,2} = 10; v_{C,3} = \cdots = 0$ **vorgegeben.**

- Bedingung C ist erfüllt für alle **x** ≥ 10 und **i**, **y**, ..., **y4** ≥ 0.
- *BIR*: 20 * (36 – 10) * 49 * 9 * 9 * 4 * 4 = 33.022.080

Veranschaulichung (4)

```
for (i = 0; i < 20; i++)
  for (x = 0; x < 36; x++)
    if (x >= 10)
      for (; x < 36; x++) ...
          for (y4 = 0; y4 < 4; y4++) {...}
    else
      for (y = 0; y < 49; y++) ...
          for (y4 = 0; y4 < 4; y4++) {
            if (4*x+vx+x4-4>35) ... }
```

BIR: *Anzahl der Ausführungen des Codes in geschweiften Klammern für* **x** ≥ 10.

Hypothetischer Code für $v_{C,2} = 10$

– Splitting-Schleife $\lambda = 2$, da *Splitting-If* in 2. Schleife steht.

– Frage: Wie oft würden *alle* hier vorkommenden *If-Statements* ausgeführt?

Zählen von *If-Statement* Ausführungen

- **#*If-Statements* nach *Splitting*** *(#Originale If´s + #Splitting-If´s)*

  $$IF_{\text{Total}} = IF_{\text{Orig}} + IF_{\text{Split}}$$

- **#Originale *If´s*** *(Totale Iterationen ohne Eingeschränkte Iterationen)*

  $$IF_{\text{Orig}} = TIR - BIR$$

- **Splitting-*If´s***

  $$IF_{\text{Split}} = \#Then\text{-}blocks + \#Else\text{-}blocks = TB + EB$$

- ***#Then-Blocks*** *(BIR ohne Iterationen von Schleifen innerhalb von λ)*

  $$TB = BIR/(\prod_{L=\lambda+1}^{N} (u_L - l_L + 1) * r_\lambda)$$

- ***#Else-Blocks*** *(#Originale If´s ohne Schleifen-Iterationen in λ)*

  $$EB = IF_{\text{Orig}}/\prod_{L=\lambda+1}^{N} (u_L - l_L + 1)$$

Veranschaulichung (5)

```
for (i = 0; i < 20; i++)
  for (x = 0; x < 36; x++)
    if (x >= 10)
      for (; x < 36; x++) ...
          for (y4 = 0; y4 < 4; y4++) {...}
    else
      for (y = 0; y < 49; y++) ...
          for (y4 = 0; y4 < 4; y4++) {
            if (4*x+vx+x4-4>35) ... }
```

IF_{Split}

IF_{Orig}

IF_{Total}

Veranschaulichung (6)

```
for (i = 0; i < 20; i++)
  for (x = 0; x < 36; x++)
    if (x >= 10)
      for (; x < 36; x++) ...
          for (y4 = 0; y4 < 4; y4++) {...}
    else
      for (y = 0; y < 49; y++) ...
          for (y4 = 0; y4 < 4; y4++) {
           if (4*x+vx+x4-4>35) ... }
```

BIR

IF_{Orig}

TIR

$$IF_{\text{Orig}} = 20 * 36 * 49 * 9 * 9 * 4 * 4 - 20 * (36 - 10) * 49 * 9 * 9 * 4 * 4$$
$$= 45.722.880 - 33.022.080 = 12.700.800$$

Veranschaulichung (7)

```
for (i = 0; i < 20; i++)
  for (x = 0; x < 36; x++)
    if (x >= 10)
      for (; x < 36; x++) ...
          for (y4 = 0; y4 < 4; y4++) {...}
    else
      for (y = 0; y < 49; y++) ...
          for (y4 = 0; y4 < 4; y4++) {
            if (4*x+vx+x4-4>35) ... }
```

$\#Then\text{-}blocks$ } IF_{Split}

$\#Else\text{-}blocks$

Veranschaulichung (8)

```
for (i = 0; i < 20; i++)
  for (x = 0; x < 36; x++)
    if (x >= 10)
      for (; x < 36; x++) ...
          for (y4 = 0; y4 < 4; y4++) {...}
    else
      for (y = 0; y < 49; y++)
        for (vx = 0; vx < 9; vx++)
          for (vy = 0; vy < 9; vy++)
            for (x4 = 0; x4 < 4; x4++)
              for (y4 = 0; y4 < 4; y4++) {
                if (4*x+vx+x4-4>35) ... }
```

$IF_{\text{Orig}}/4/4/9/9/49$

$IF_{\text{Orig}}/4/4/9/9$

$IF_{\text{Orig}}/4/4/9$

$IF_{\text{Orig}}/4/4$

$IF_{\text{Orig}}/4$

IF_{Orig}

$$\#\textit{Else-blocks} = IF_{\text{Orig}}/(49 * 9 * 9 * 4 * 4)$$
$$= 12.700.800/63.504 = 200$$

Veranschaulichung (9)

```
for (i = 0; i < 20; i++)
  for (x = 0; x < 36; x++)
    if (x >= 10)
      for (; x < 36; x++) ...
        for (y = 0; y < 49; y++)
          for (vx = 0; vx < 9; vx++)
            for (vy = 0; vy < 9; vy++)
              for (x4 = 0; x4 < 4; x4++)
                for (y4 = 0; y4 < 4; y4++) {
                  ...}
    else
      ...
```

$BIR/4/4/9/9/49/26$

$BIR/4/4/9/9/49$

BIR

analog zur vorigen Folie

$$\#\mathit{Then\text{-}blocks} = BIR/(26 * 49 * 9 * 9 * 4 * 4)$$
$$= 33.022.080/1.651.104 = 20$$

Veranschaulichung (10)

```
for (i = 0; i < 20; i++)
  for (x = 0; x < 36; x++)
    if (x >= 10)
      for (; x < 36; x++) ...
          for (y4 = 0; y4 < 4; y4++) {...}
    else
      for (y = 0; y < 49; y++) ...
          for (y4 = 0; y4 < 4; y4++) {
            if (4*x+vx+x4-4>35) ... }
```

IF_{Split} (then-branch)

IF_{Orig} (else-branch)

IF_{Total} (whole loop nest)

$$
\begin{aligned}
IF_{\text{Total}} &= IF_{\text{Orig}} + IF_{\text{Split}} = IF_{\text{Orig}} + \#\mathit{Then\text{-}Blocks} + \#\mathit{Else\text{-}Blocks} \\
&= 12.700.800 + 20 + 200 \\
&= 12.701.020
\end{aligned}
$$

Erzeugen der Werte $v_{C,L}$

Bisher Erreichtes

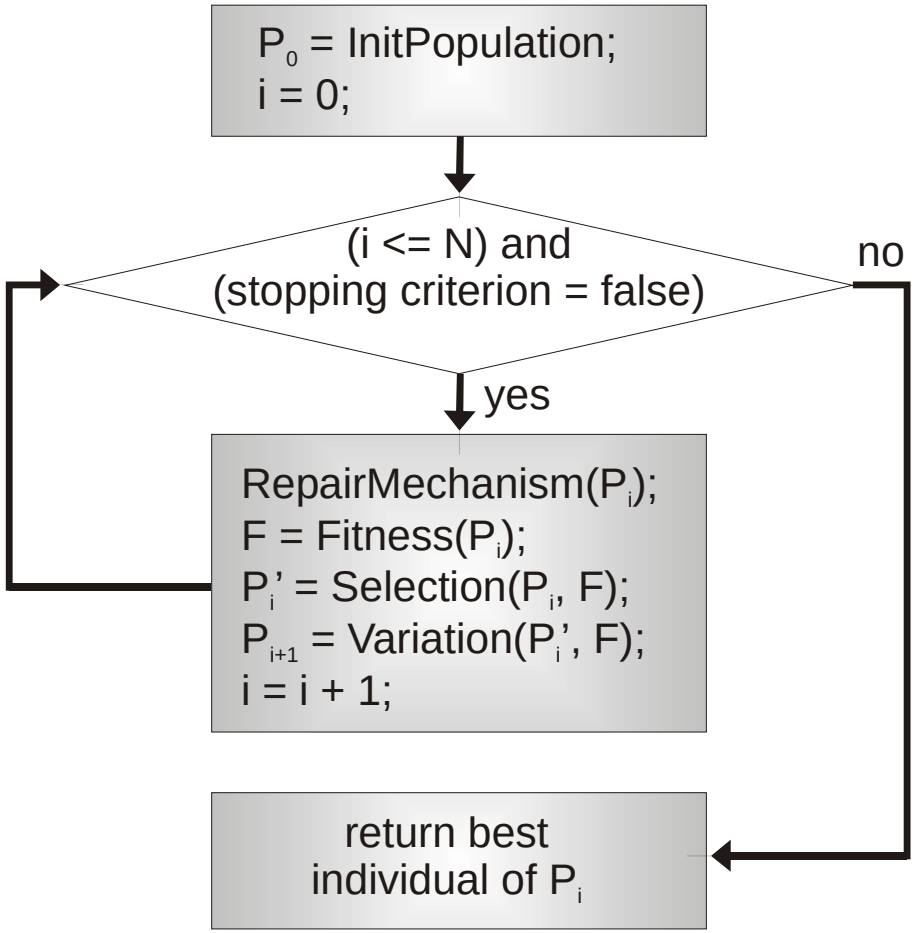
– Für eine Bedingung C und gegebene Werte $v_{C,L}$ kann berechnet werden, wie viele *If-Statements* nach *Splitting* anhand von $v_{C,L}$ ausgeführt werden würden.

Sehr schön, aber...

... wer erzeugt gute Werte für $v_{C,L}$?

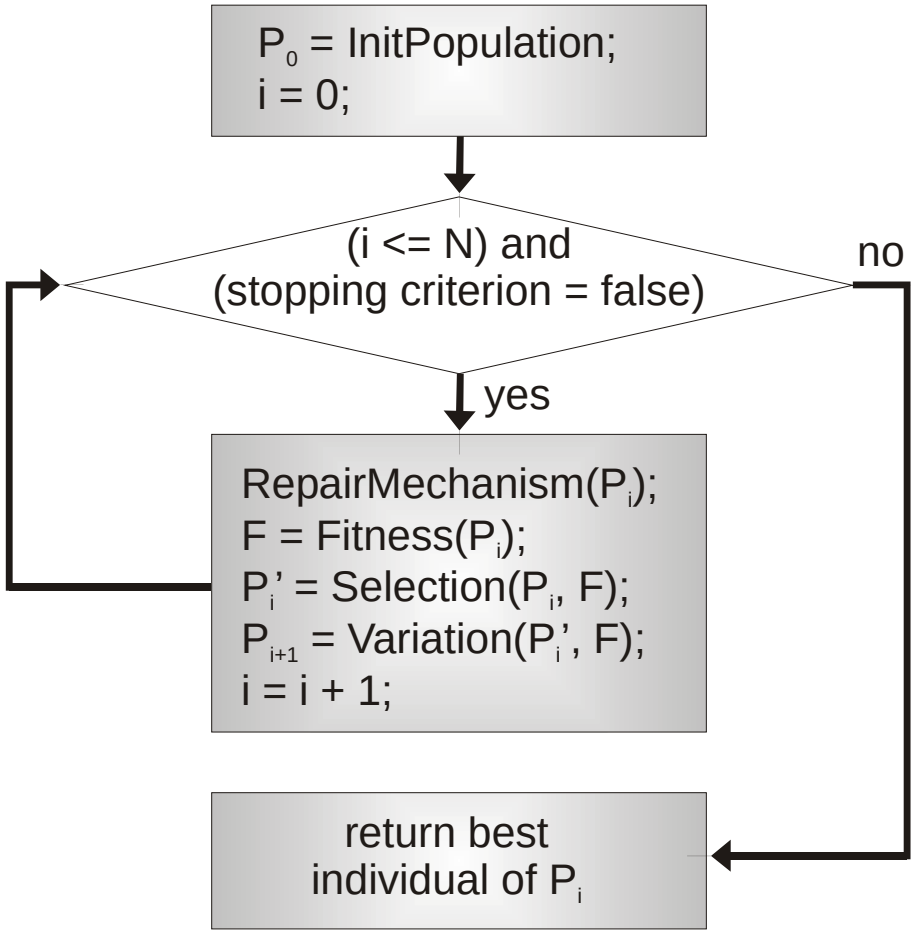
☞ **Ein Genetischer Algorithmus**

Ablauf Genetischer Algorithmen (1)

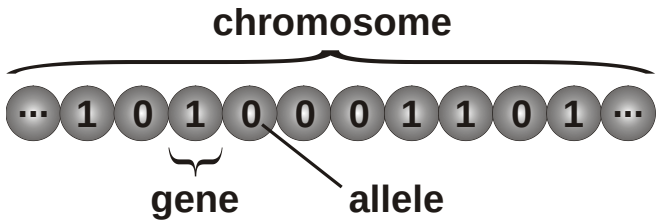


- Anlehnung an natürliche Evolution, *„Survival of the fittest“*
- Optimierung in Schleife *i* = 0, 1, ...
- Jede Iteration *i* verwaltet Population P_i; eine Population besteht aus vielen *Individuen*
- Ein Individuum repräsentiert eine Lösung für das modellierte Optimierungsproblem

Ablauf Genetischer Algorithmen (2)

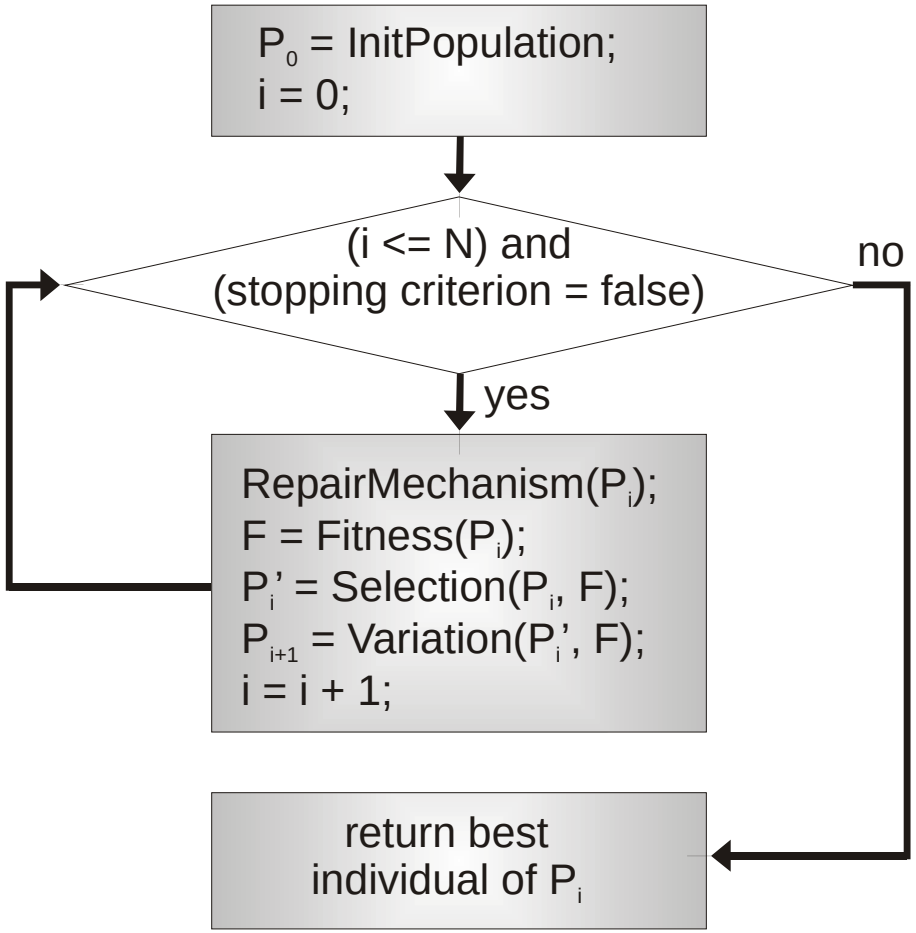


– Die Datenstruktur eines Individuums heißt *Chromosom*.



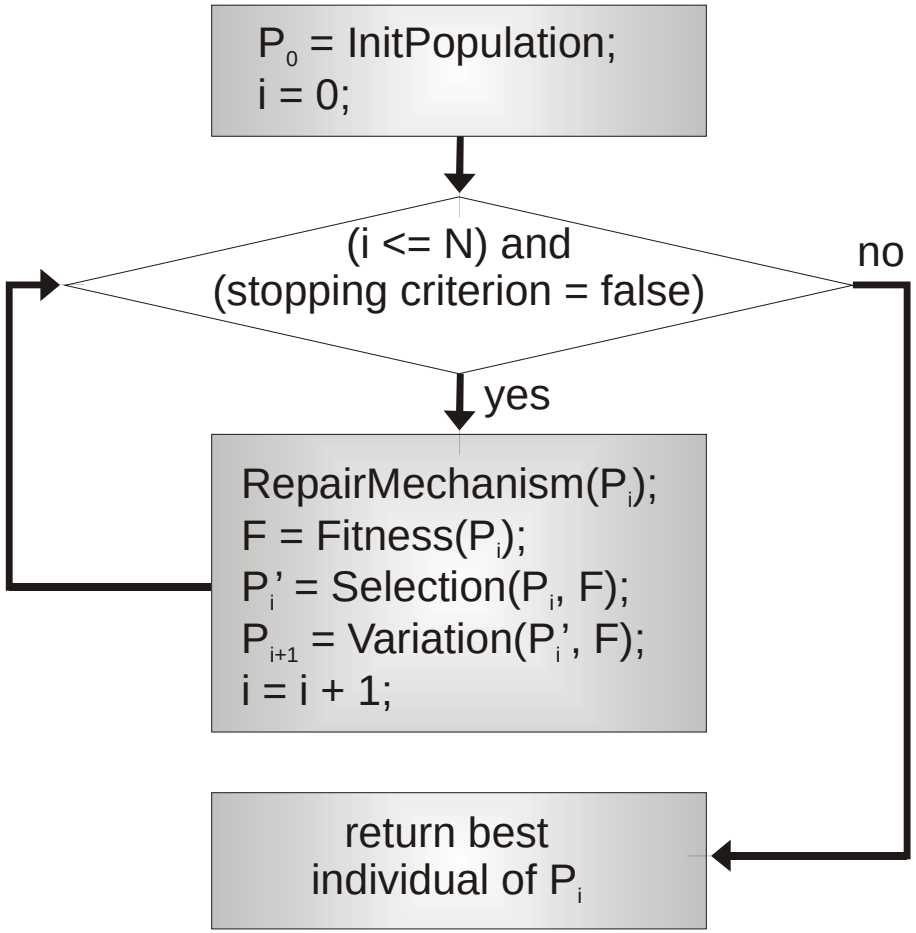
– Ein Chromosom besteht aus vielen *Genen*, die Daten speichern.
– Ein konkreter in einem Gen gespeicherter Wert heißt *Allel*.

Ablauf Genetischer Algorithmen (3)

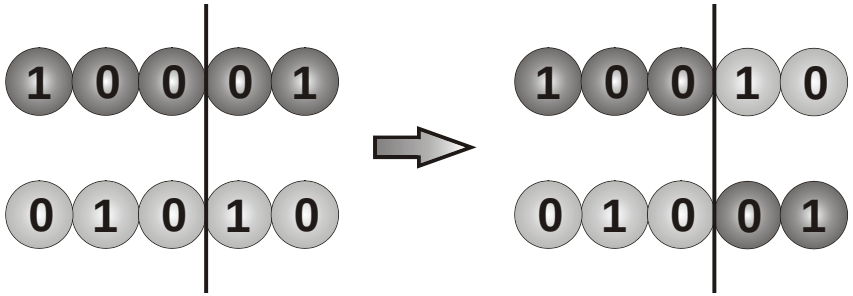


- Eine Fitness-Funktion berechnet für jedes Individuum in P_i dessen *Fitness*.
- Aus P_i wird eine Teilmenge P_i' mit höchster / niedrigster Fitness ausgewählt (*Selektion*, abhängig ob Minimierungs- oder Maximierungsproblem).
- P_i' wird durch zufällig generierte neue Individuen zur nächsten Population P_{i+1} ergänzt *(Variation)*

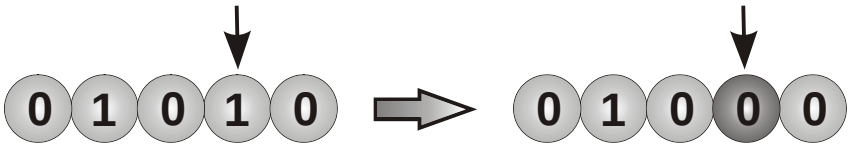
Ablauf Genetischer Algorithmen (4)



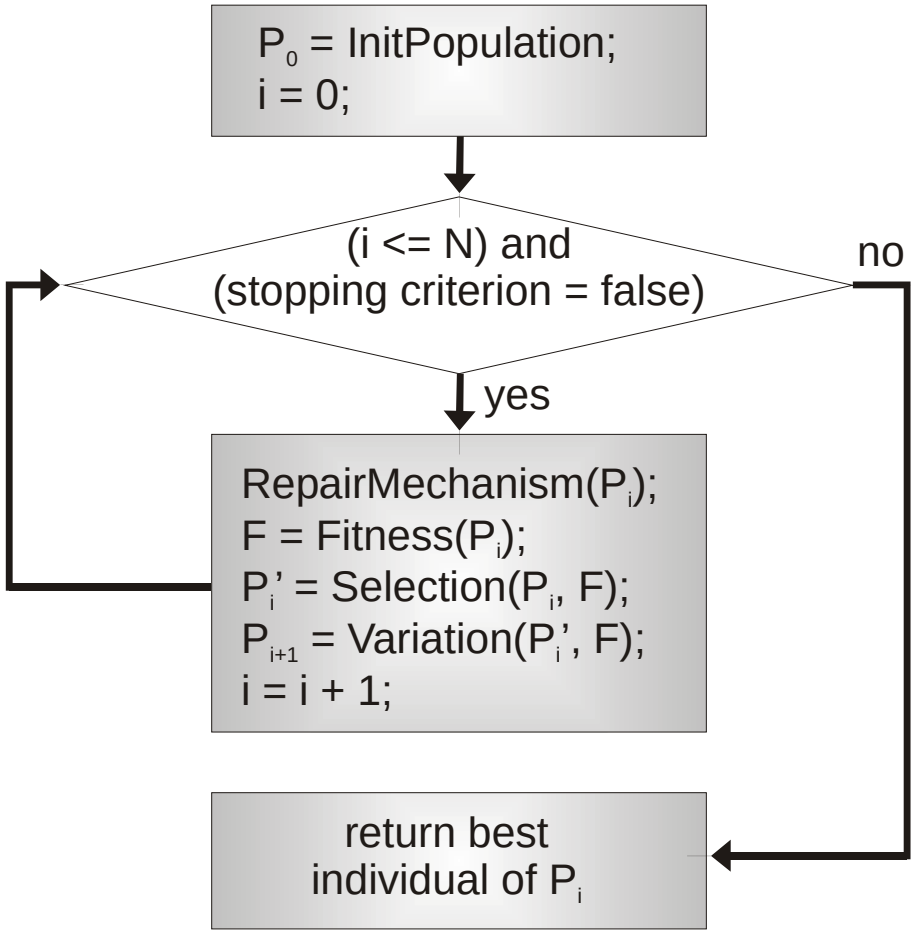
- Variation beruht auf zwei Genetischen Operatoren:
- *Crossover:*



- *Mutation:*



Ablauf Genetischer Algorithmen (5)



- Wegen zufälliger Variation kann P_i Individuen enthalten, die keine gültige Lösung repräsentieren. Diese müssen *repariert* werden.
- Abbruch des GAs, wenn
  - max. *N* Iterationen erreicht,
  - beste erreichte Fitness über *y* Iterationen unverändert,
  - ...
- Rückgabe des Individuums mit bester Fitness aus letzter Population als Endergebnis.

Der Genetische Algorithmus zur *Condition Optimization*

Chromosomale Repräsentation

- Bei *N* verschachtelten Schleifen hat jedes Chromosom *N* Gene
- Jedes Gen stellt eine *integer*-Zahl dar
- Gen *L* repräsentiert zu optimierenden Wert $v_{C,L}$
- Wertebereich jedes Gens *L* beschränkt auf $[l_L, u_L]$

Fitness

- Die Fitness eines Individuums = IF_{Total}
- Ungültige Individuen $(v_{C,1}, \ldots, v_{C,N})$ repräsentieren Iterationen, in denen Bedingung C nicht immer erfüllt ist
- Ungültige Individuen erhalten sehr schlechte Fitness

Ergebnis von *Condition Optimization*

Eingabe der *Condition Optimization*

– Lineare Bedingung C

– Schleifengrenzen $[l_L, u_L]$

Ausgabe des Genetischen Algorithmus

– Werte $(v_{C,1}, \ldots, v_{C,N})$ des Individuums mit bester Fitness

Ausgabe der *Condition Optimization*

– Polytop

$$P'_C = \left\{ (x_1, \ldots, x_N) \in \mathbb{Z}^N \mid \forall \text{ Schleifen } L: \begin{array}{l} l_L \leq x_L \leq u_L, \\ x_L \geq v_{C,L} \text{ falls } c_L > 0, \\ x_L \leq v_{C,L} \text{ falls } c_L < 0 \end{array} \right\}$$

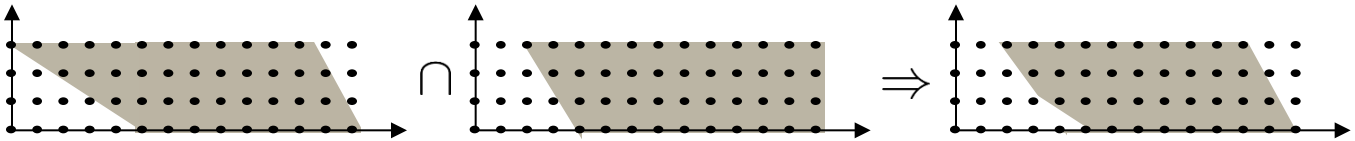
Phase 3 – *Search Space Generation*

Gegeben: *If-Statements*, Bedingungen & Polytope

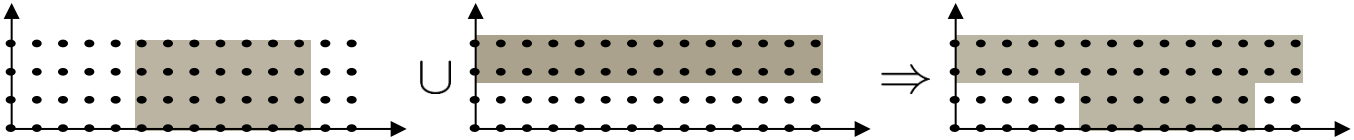
$$\mathrm{IF}_i = (C_{i,1} \otimes C_{i,2} \otimes \cdots \otimes C_{i,n}), \otimes \in \{\&\&, ||\} \qquad \forall C_{i,j} \rightsquigarrow P_{i,j}$$

Konstruktion von EVPs P_i für jedes *If-Statement* IF_i

– if $C_{i,j-1}$ && $C_{i,j}$:



– if $C_{i,j-1}$ || $C_{i,j}$:



Konstruktion einer Globalen EVP *(Global Search Space)*

EVP *G* modelliert Iterationsraum, in dem *alle* *If-Statements* wahr sind

– $G = \bigcap P_i$

Phase 3 – Struktur der EVP *G*

Konsequenz der Verwendung des $\cup$ Operators

- *G* ist Endliche Vereinigung von Polyedern
- $G = R_1 \cup R_2 \cup \cdots \cup R_M$
- Interpretation:

  Jedes einzelne Polytop $R_r \rightsquigarrow$

  *Region im Iterationsraum, für die **alle** If-Statements wahr sind.*

Global Search Space für MPEG-Code

- $\mathrm{IF}_1 =$ `4*x+x4<0 || 4*x+x4>35 || 4*y+y4<0 || 4*y+y4>48`

  $P_{1,1} = \emptyset, P_{1,2} = \{\mathrm{x} \geq 9\}, P_{1,3} = \emptyset, P_{1,4} = \{\mathrm{y} \geq 13\}$

  $P_1 = \{\mathrm{x} \geq 9\} \cup \{\mathrm{y} \geq 13\}$

- $\mathrm{IF}_2 =$ `4*x+vx+x4<4 || 4*x+vx+x4>39 || 4*y+vy+y4<4 || 4*y+vy+y4>52`

  $P_{2,1} = \{\mathrm{x} = 0 \ \wedge \ \mathrm{vx} = 0\}, P_{2,2} = \{\mathrm{x} \geq 10\},$

  $P_{2,3} = \{\mathrm{y} = 0 \ \wedge \ \mathrm{vy} = 0\}, P_{2,4} = \{\mathrm{y} \geq 14\}$

  $P_2 = \{\mathrm{x} = 0 \ \wedge \ \mathrm{vx} = 0\} \cup \{\mathrm{x} \geq 10\} \cup \{\mathrm{y} = 0 \ \wedge \ \mathrm{vy} = 0\} \cup \{\mathrm{y} \geq 14\}$

- $G = P_1 \cap P_2 =$

  $\{\mathrm{x} = 0 \ \wedge \ \mathrm{vx} = 0 \ \wedge \ \mathrm{y} \geq 13\} \ \cup \ \{\mathrm{x} \geq 10\} \cup$

  $\{\mathrm{y} = 0 \ \wedge \ \mathrm{vy} = 0 \ \wedge \ \mathrm{x} \geq 9\} \ \cup \ \{\mathrm{y} \geq 14\}$

Global Search Space & Splitting-If (1)

- $G = \mathrm{IF}_1 \cap \mathrm{IF}_2 =$
  $\{\mathtt{x} = 0 \ \wedge \ \mathtt{vx} = 0 \ \wedge \ \mathtt{y} \geq 13\} \ \cup \ \{\mathtt{x} \geq 10\} \cup$
  $\{\mathtt{y} = 0 \ \wedge \ \mathtt{vy} = 0 \ \wedge \ \mathtt{x} \geq 9\} \ \cup \ \{\mathtt{y} \geq 14\}$

Direkte Übersetzung von *G* in *Splitting-If*

```
if ( (x == 0 && vx == 0 && y >= 13) || (x >= 10) ||
     (y == 0 && vy == 0 && x >= 9) || (y >= 14) )
```

Keine gute Lösung

- ☞ Dieses *Splitting-If* müsste in **vy**-Schleife platziert sein (3.-innerste!)
- ☞ Führt zu 10.103.760 Ausführungen von *If-Statements*

Global Search Space & Splitting-If (2)

- $G = \mathrm{IF}_1 \cap \mathrm{IF}_2 =$
  $\{\mathtt{x} = 0 \ \wedge \ \mathtt{vx} = 0 \ \wedge \ \mathtt{y} \geq 13\} \ \cup \ \{\mathtt{x} \geq 10\} \cup$
  $\{\mathtt{y} = 0 \ \wedge \ \mathtt{vy} = 0 \ \wedge \ \mathtt{x} \geq 9\} \ \cup \ \{\mathtt{y} \geq 14\}$

Alternativ: Nur Teil-Polytope R_r von *G* für *Splitting-If* benutzen.
Legal, da jedes Teil-Polytop R_r für sich bereits alle *If-Statements* erfüllt.

```
if ( x >= 10 )
```

- Dieses Teil-Polytop ist auch keine gute Lösung

  ☞Führt zu 25.401.820 Ausführungen von *If-Statements*

```
if ( ( x >= 10 ) || ( y >= 14 ) )
```

- Diese Teil-Polytope sind eine gute Lösung

  ☞Führen zu 7.261.120 Ausführungen von *If-Statements*

Phase 4 – *Search Space Exploration*

GA: Wählt Regionen aus $G = R_1 \cup R_2 \cup \cdots \cup R_M$ aus

– Individuum: Bitvektor, der die ausgewählten Regionen codiert

$$\text{Individuum } I = (I_1, \ldots, I_M), I_r = \begin{cases} 1 & \text{wenn Region } R_r \text{ gewählt,} \\ 0 & \text{sonst} \end{cases}$$

– Fitness-Funktion berechnet wieder
#{*If-Statement* Ausführungen} nach *Loop Nest Splitting*

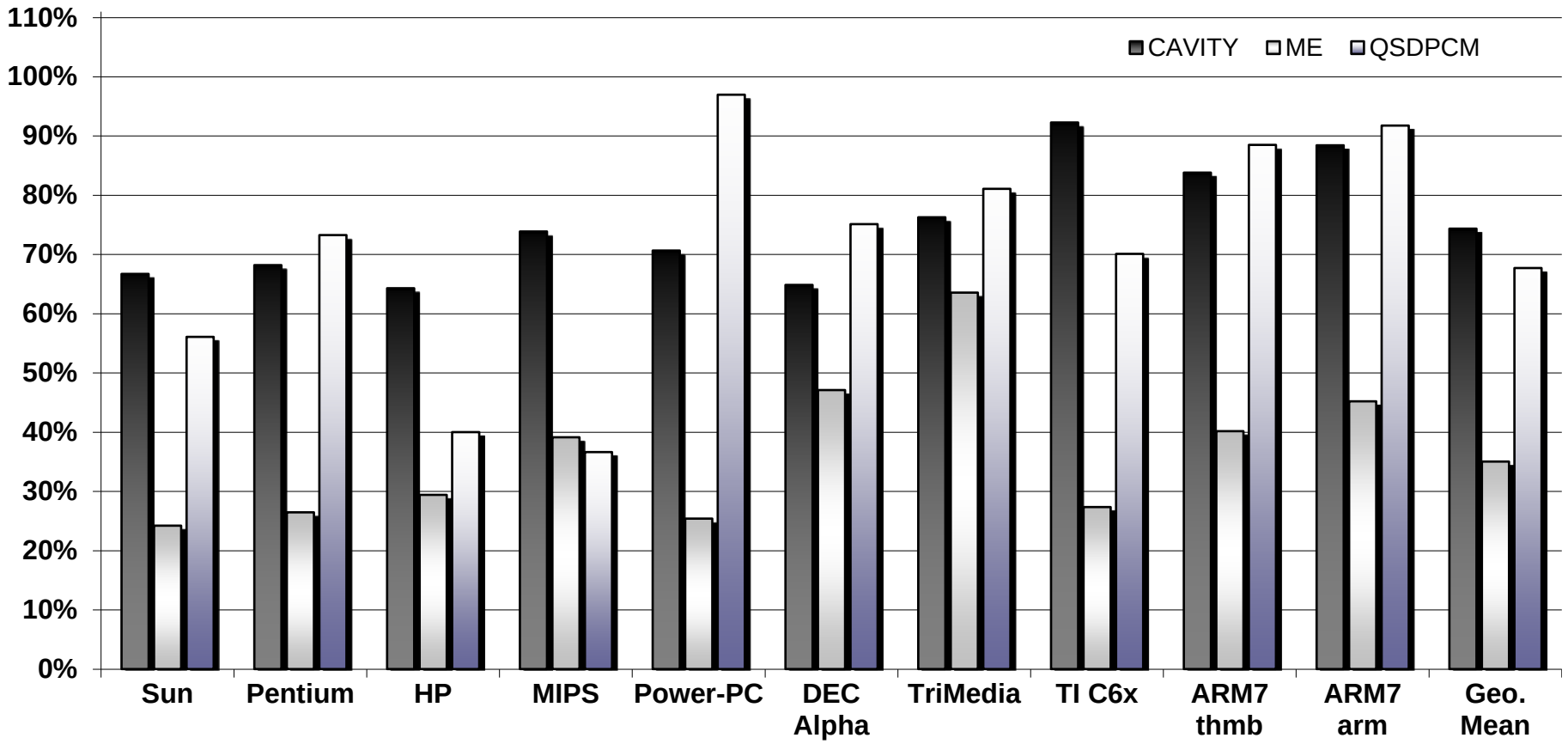
– Fitness-Funktion wird durch GA minimiert

Resultierendes *Splitting-If Statement*

– So weit wie möglich in äußerster Schleife platziert

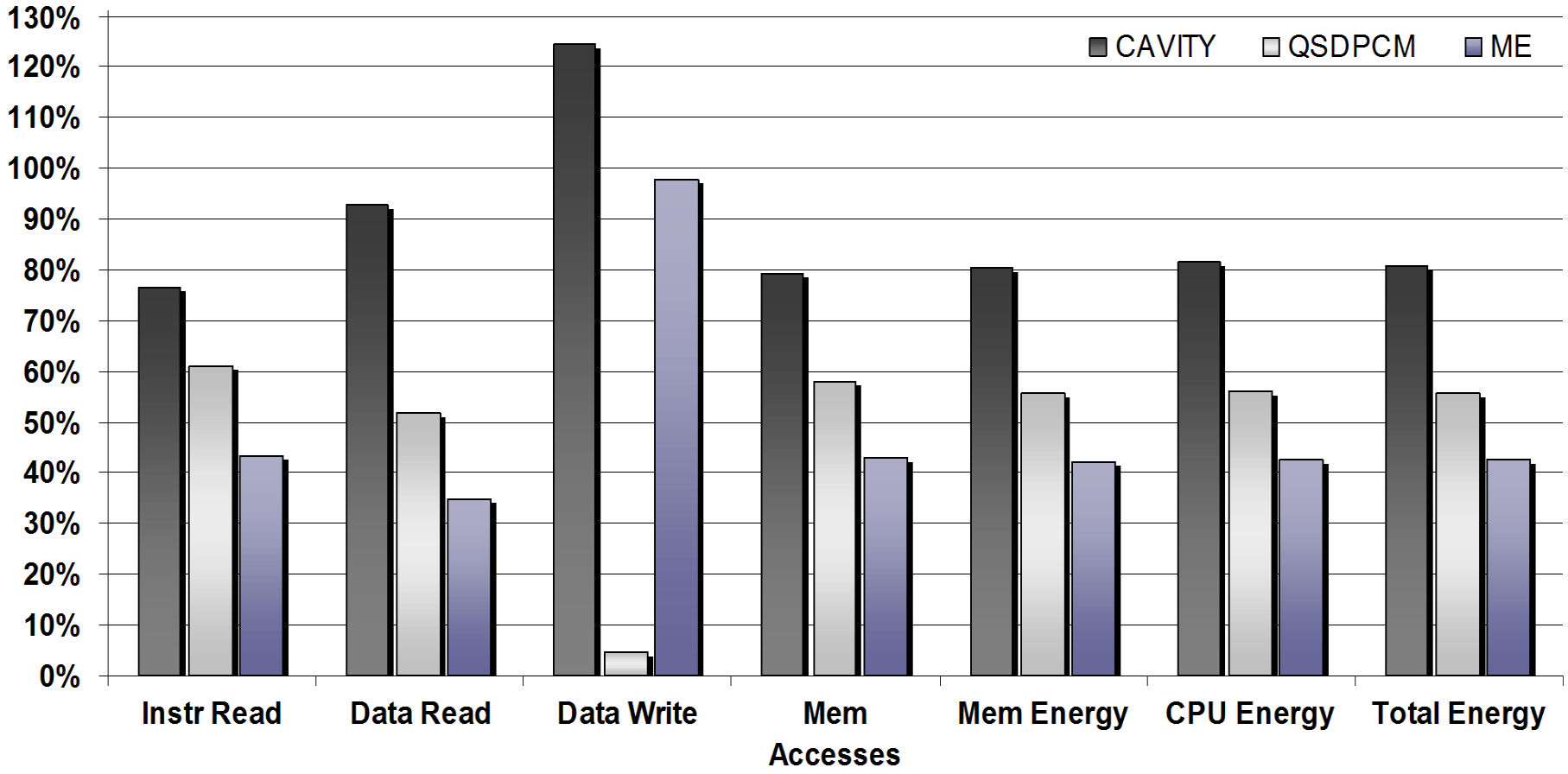
– Enthält alle Bedingungen und Operatoren wie durch die gewählten Regionen R_r vorgegeben

Relative Laufzeiten nach *Loop Nest Splitting*



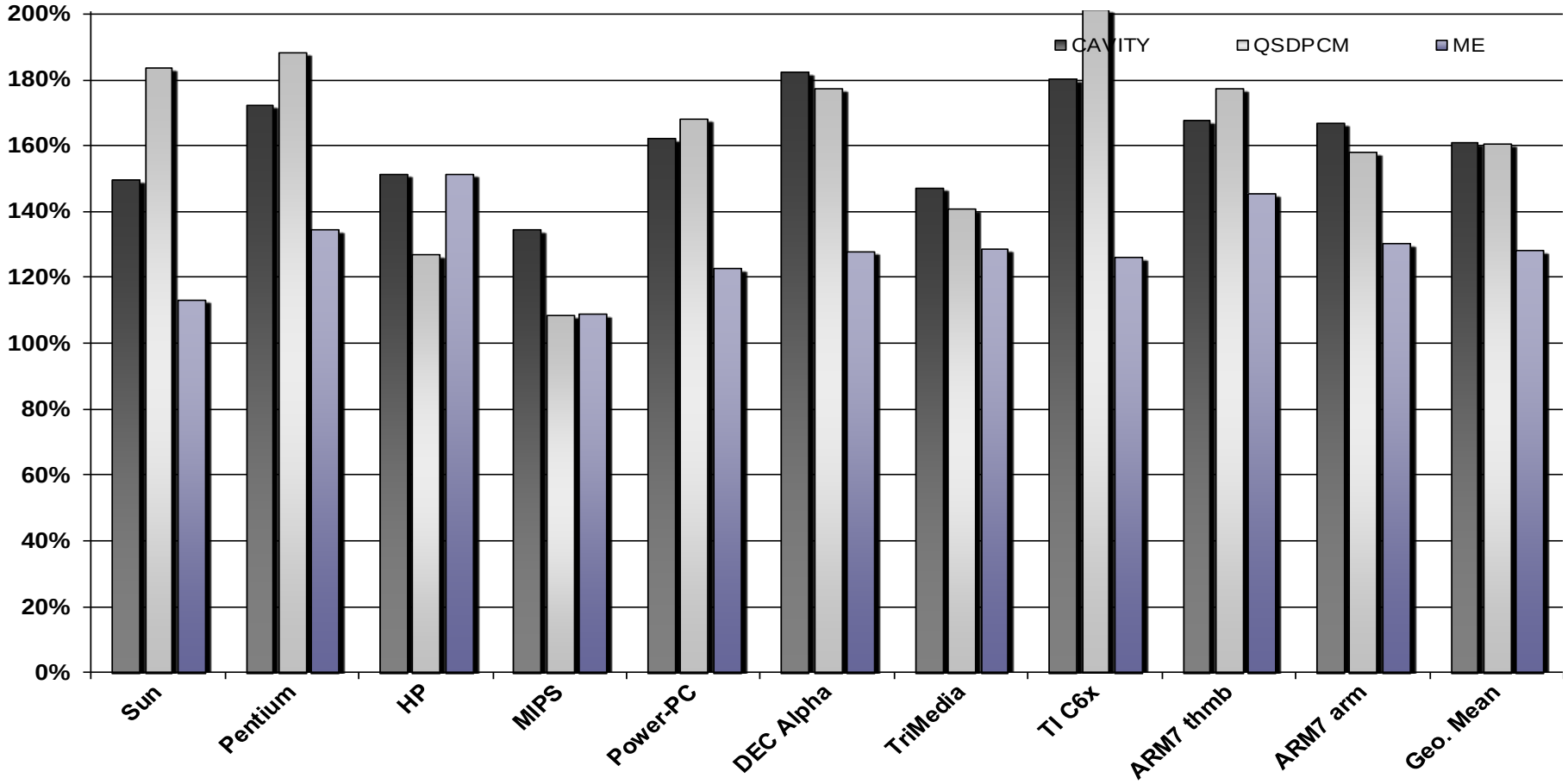
100% = Laufzeit der Benchmarks ohne Loop Nest Splitting

Relativer Energieverbrauch (ARM7) nach *LNS*



100% = Werte der Benchmarks ohne Loop Nest Splitting

Relative Codegröße nach *Loop Nest Splitting*



100% = Größe der Benchmarks ohne Loop Nest Splitting

Literatur

Loop Nest Splitting

– H. Falk, P. Marwedel. *Control Flow driven Splitting of Loop Nests at the Source Code Level*. DATE Conference, München, 2003.

– H. Falk. *Control Flow Optimization by Loop Nest Splitting at the Source Code Level*. Universität Dortmund, Forschungsbericht Nr. 773, Dortmund 2003.

Zusammenfassung

Non-Compiler-Optimierungen

– *Postpass* nach Compiler, z.B. auf Linker-Niveau
– *Prepass* vor Compiler, auf Quellcode-Niveau

Loop Nest Splitting

– Kontrollfluss-Optimierung in Datenfluss-dominierten Multimedia-Anwendungen
– Polytope zur Modellierung linearer Bedingungen und Schleifen
– Genetische Algorithmen zur Optimierung der Polytop-Modelle
– Enorme Einsparungen in ACET und Energie *(und WCET)*, aber z.T. beträchtliche Code-Vergrößerung