TUHH
Hamburg University of Technology

Institute of Embedded Systems

Compilers for Embedded Systems

Summer Term 2022

Heiko Falk

Institute of Embedded Systems
Electrical Engineering, Computer Science and Mathematics
Hamburg University of Technology

TUHH
Hamburg University of Technology

Institute of
Embedded Systems

Chapter 4

Pre-Pass Optimizations

Outline

1. Introduction & Motivation
2. Compilers for Embedded Systems – Requirements & Dependencies
3. Internal Structure of Compilers
4. **Pre-Pass Optimizations**
5. HIR Optimizations and Transformations
6. Code Generation
7. LIR Optimizations and Transformations
8. Register Allocation
9. WCET-Aware Compilation
10. Outlook

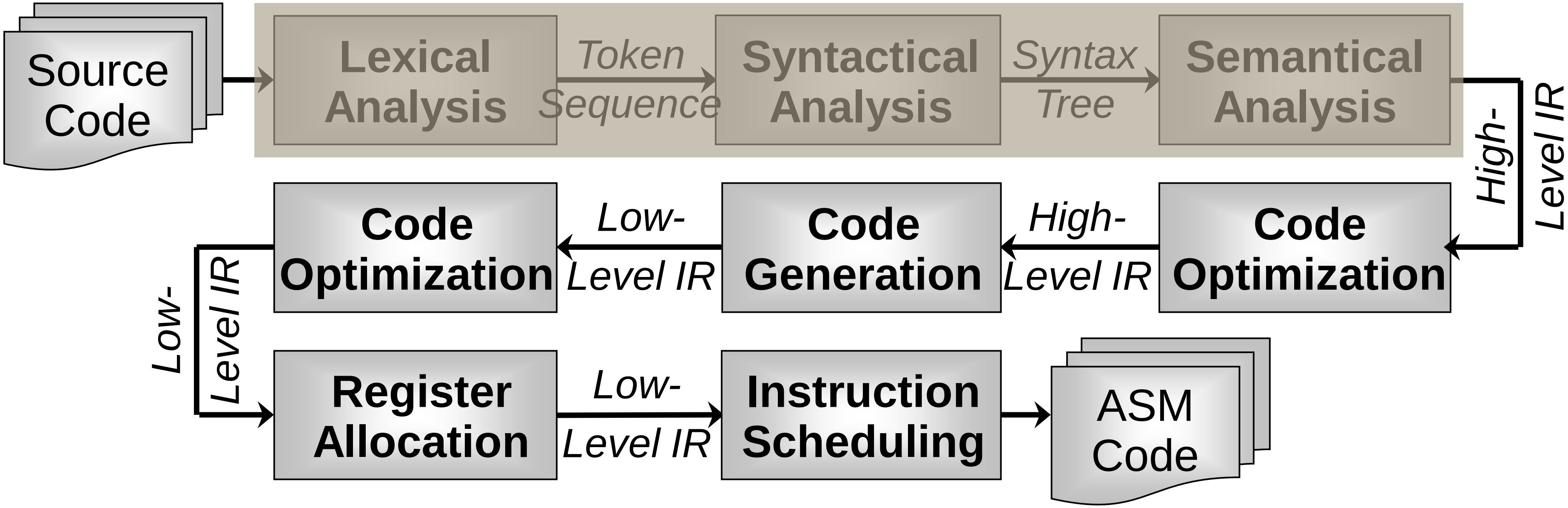
Chapter Contents

4. Pre-Pass Optimizations

- Motivation of Pre-Pass Optimizations
- Loop Nest Splitting
    - Embedded Multimedia: MPEG 4 Motion Estimation
    - Workflow of Loop Nest Splitting
    - Condition Satisfiability
    - Condition Optimization & Genetic Algorithms
    - Search Space Generation
    - Search Space Exploration
    - Results (ACET, Energy, Code Size)

Motivation of Pre-Pass Optimizations (1)

☞ ***Retrospect: Structure of an Optimizing Compiler with 2 IRs:***



Source Code

Lexical Analysis

Token Sequence

Syntactical Analysis

Syntax Tree

Semantical Analysis

High-Level IR

Code Optimization

High-Level IR

Code Generation

Low-Level IR

Code Optimization

Low-Level IR

Register Allocation

Low-Level IR

Instruction Scheduling

ASM Code

Question: May only the compiler optimize code?

Motivation of Pre-Pass Optimizations (2)

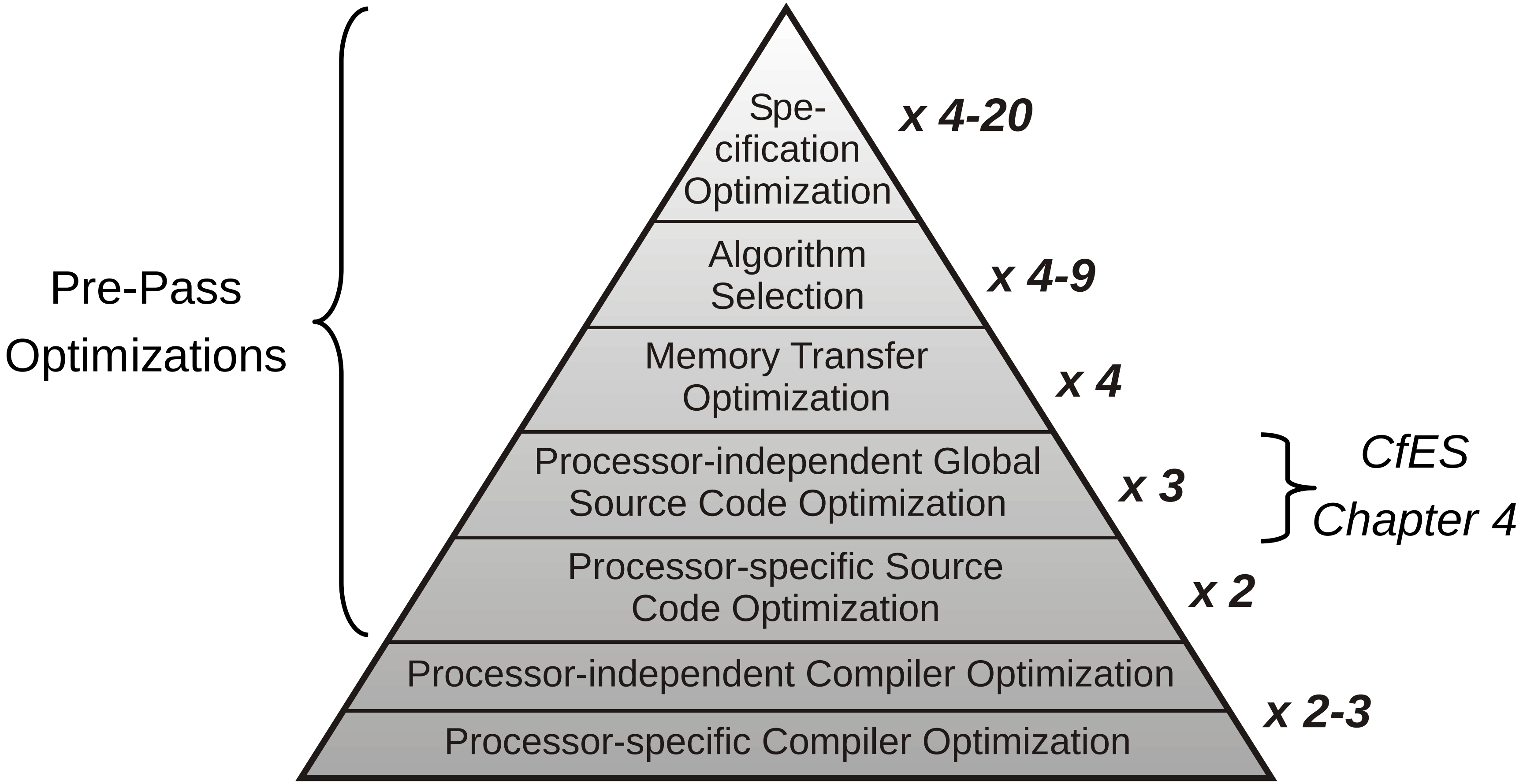
Optimizations outside a compiler are called

- *Post-pass* if they take place *after* the compiler
- *Pre-pass* if they take place *before* the compiler

Advantages of Pre-Pass Optimizations

- Source code transformations more easily comprehensible.
- Allow for manually “playing” with an optimization technique before a laborious implementation.
- Independent of the actual compiler due to source code-level; principally applicable for each individual compiler of the source language.
- Due to source code-level independent of the actual target architecture; principally applicable for arbitrary architectures.

Abstraction Levels of Optimizations


Pre-Pass Optimizations
Spe-cification Optimization
x 4-20
Algorithm Selection
x 4-9
Memory Transfer Optimization
x 4
Processor-independent Global Source Code Optimization
x 3
CfES Chapter 4
Processor-specific Source Code Optimization
x 2
Processor-independent Compiler Optimization
x 2-3
Processor-specific Compiler Optimization

Chapter Contents

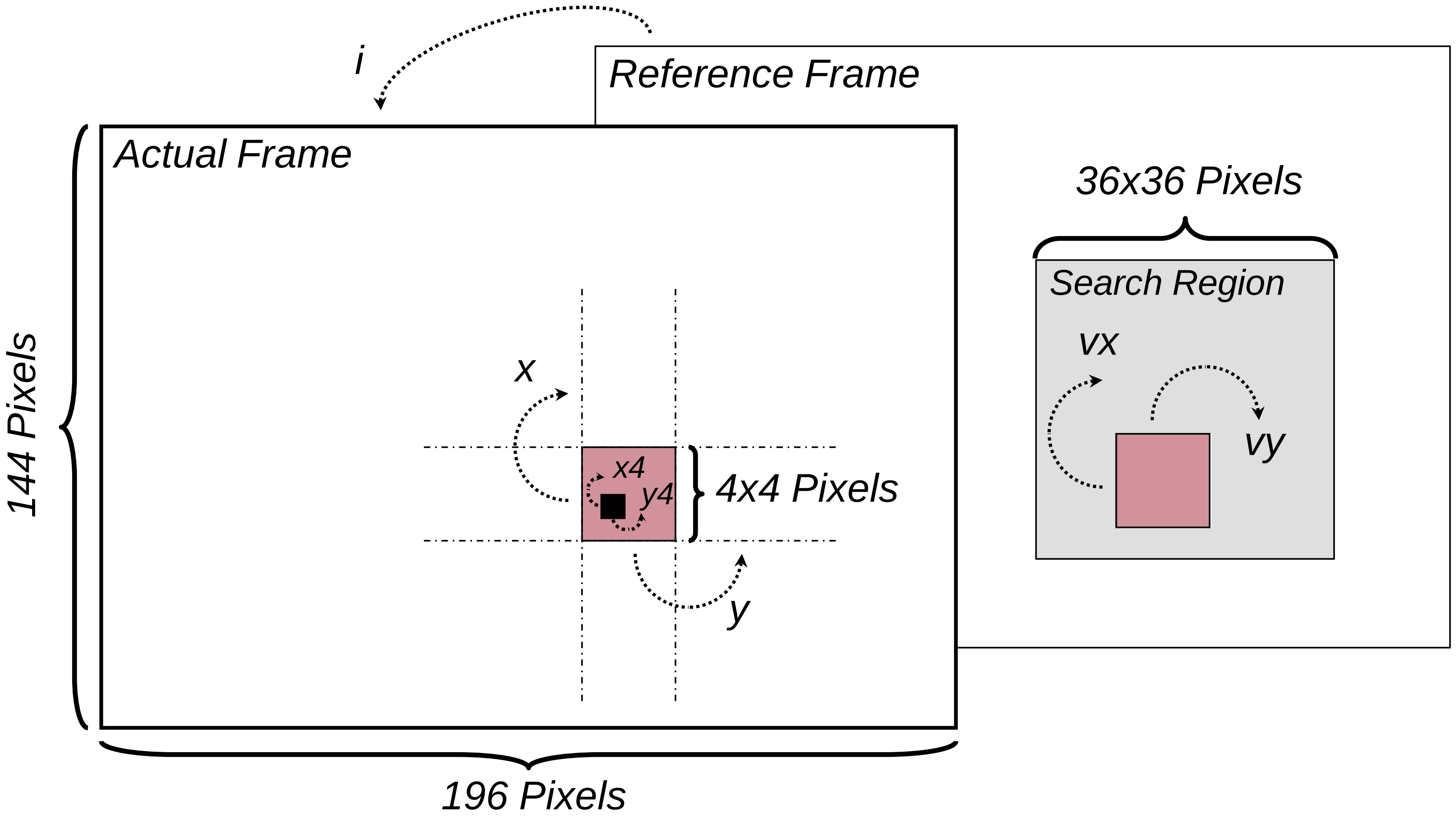
4. **Pre-Pass Optimizations**
   - Motivation of Pre-Pass Optimizations
   - Loop Nest Splitting
     - Embedded Multimedia: MPEG 4 Motion Estimation
     - Workflow of Loop Nest Splitting
     - Condition Satisfiability
     - Condition Optimization & Genetic Algorithms
     - Search Space Generation
     - Search Space Exploration
     - Results (ACET, Energy, Code Size)

Application Domain of Loop Nest Splitting

Embedded Multimedia Applications

- *Data flow dominated*, i.e., get large amounts of data as inputs, compute large volumes of data as output
  (in contrast to control flow dominated control applications).
- Largest part of the run-time consumed by (deeply) *nested loops*
- *Simple loop structures* with statically known or analyzable lower and upper bounds
- Manipulation of large, multi-dimensional arrays
- Typical example: Streaming applications like MPEG4

Example: MPEG4 Motion Estimation



Source Code 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; }
```

Observations

When Compiling and Executing this Source Code

– Execution of 91,445,760 if-statements in total

– Very irregular control flow due to if-statements

– Additional arithmetical overhead:
Multiplications, additions, comparisons, logical or, ...

☞ *Performance of this code limited by control flow, and not by the computation of motion vectors!*

Loop Nest Splitting

Automated Loop & If-Statement Analysis

– **`x`**, **`y`**, **`x4`** and **`y4`** never carry values such that conditions **`4*x+x4<0`** and **`4*y+y4<0`** are ever true.

☞ *Conditions can be replaced by constant truth value ‘0’.*

– For **`x`** ≥ 10 or **`y`** ≥ 14, both if-statements are always satisfied so that their then-parts are always executed.

☞ *For more than 92% of all executions of the innermost* **`y4`***-loop, both if-statements are satisfied.*

Source Code after 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++)             // Second y-loop
          for (vx = 0; vx < 9; vx++) ... {    // No
            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)   // Old
            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; }
```

Optimized Code Structure

Splitting-If

– Whenever the condition of the splitting-if is satisfied, the conditions of all original if-statements are automatically also satisfied.

☞ Then-part of the splitting-if thus contains no original if-statements any more, but only their then-parts.

– If splitting-if is not satisfied, no safe statement about the conditions of the original if-statements is possible.

☞ Else-part of the splitting-if contains all original if-statements in order to keep the code correct.

Why Second y-Loop?

Intuitive 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 execution for
each individual **y** ∈ [14, 48]

```
y = 16
```

Optimized 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 execution for
all **y** ∈ [14, 48] *altogether*

```
y = 16
```

Phases of Loop Nest Splitting

- ***Condition Satisfiability:*** Finds individual conditions in if-statements that are always or never satisfied.

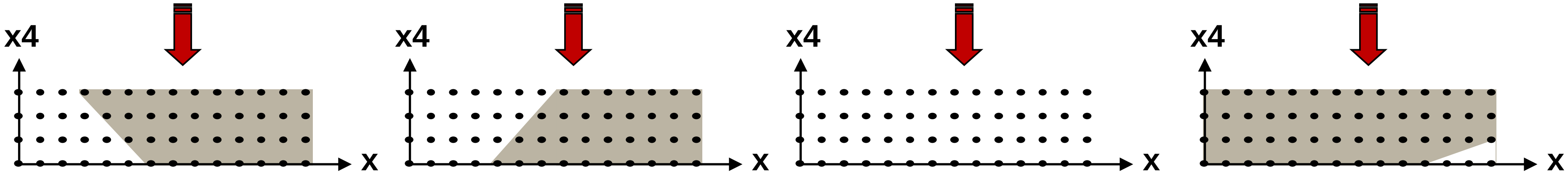
- ***Condition Optimization:*** For each single condition *C*, find a "simpler" condition *C'* such that $C' \Rightarrow C$ holds (whenever *C'* is true, *C* is also true).

- ***Search Space Generation:*** Combine all single conditions *C'* to one data structure *G* that models all if-statements including their structure (`&&`, `||`).

- ***Search Space Exploration:*** Using *G*, determine a condition for the splitting-if that minimizes the number of totally executed if-statements.

Workflow of Loop Nest Splitting (1)

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



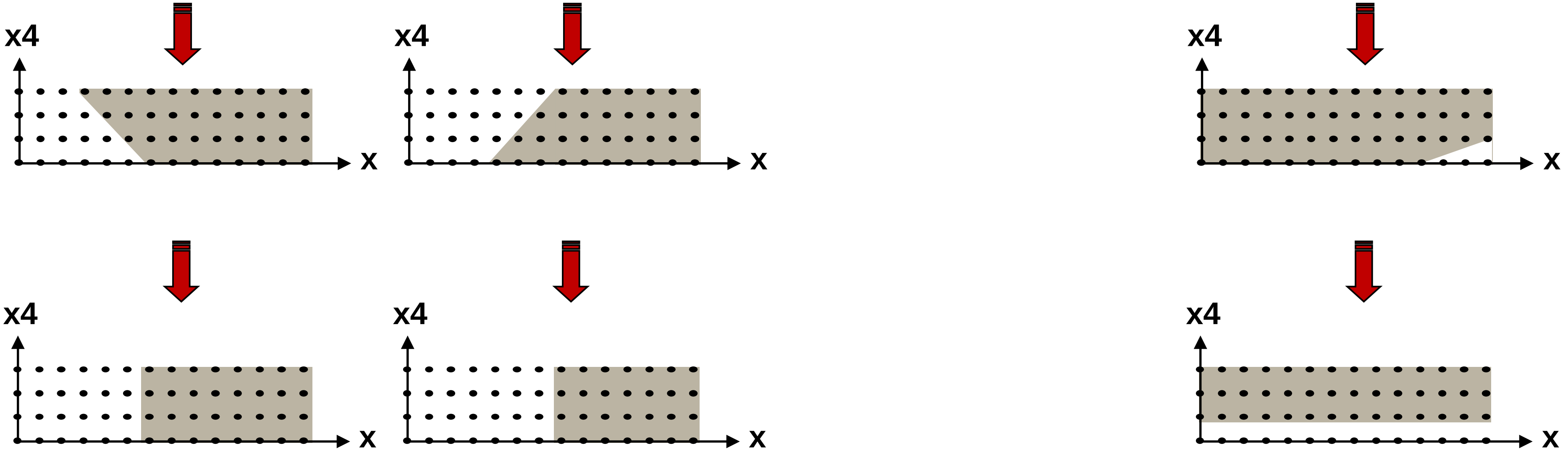
1 - Condition Satisfiability

☞ ***Note:***
This example does not correspond to the MPEG4 code from the previous slides!

– Assumed loop bounds:
$0 \le \mathbf{x} \le 13$
$0 \le \mathbf{x4} \le 3$

Workflow of Loop Nest Splitting (2)

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

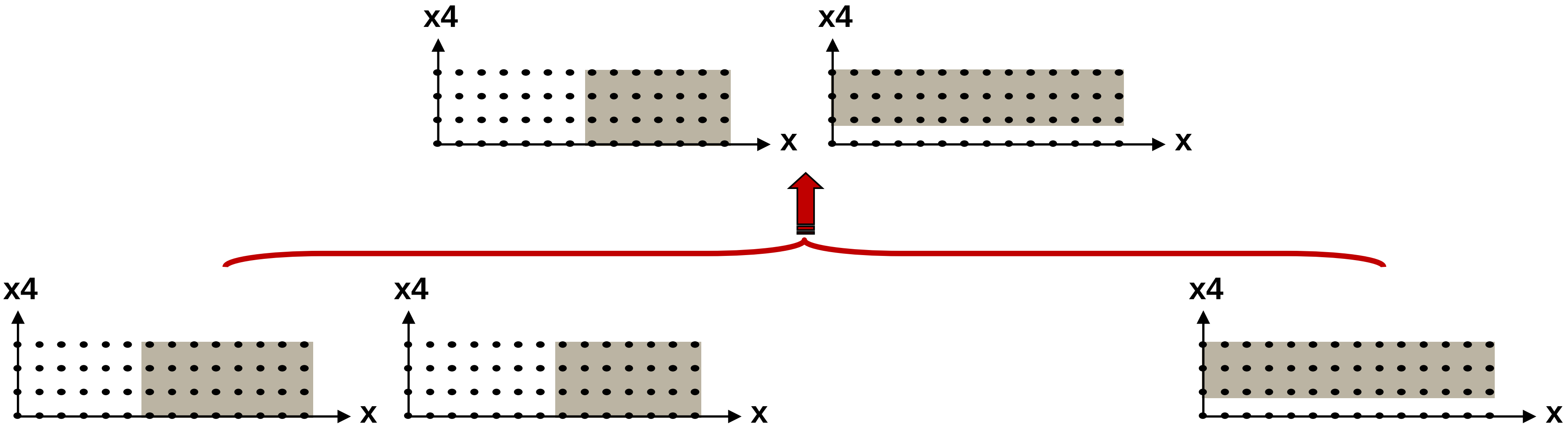


1 - Condition Satisfiability

2 - Condition Optimization

Workflow of Loop Nest Splitting (3)

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



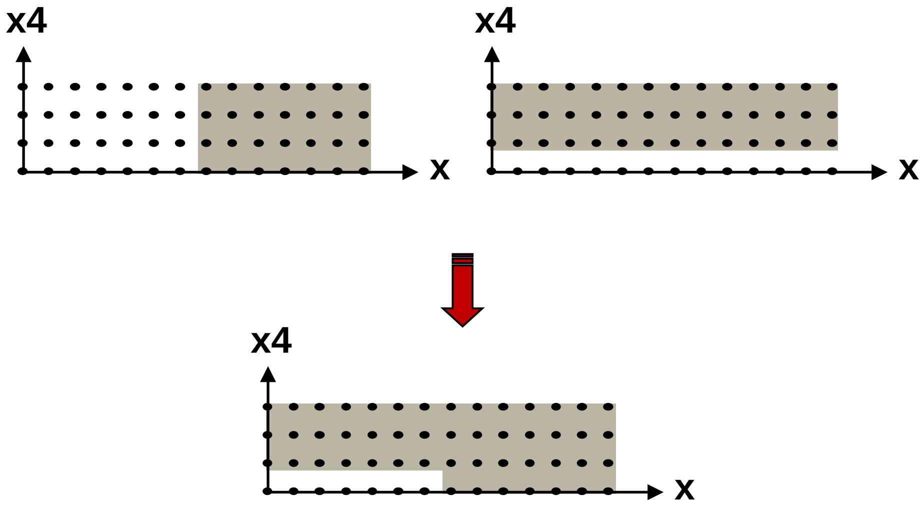
1 - Condition Satisfiability

2 - Condition Optimization

3 - Search Space Generation

Workflow of 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

Prerequisites

Bounds of Loops L

- All lower and upper bounds (l_L, u_L) are constant

If-Statements

- Sequence of loop-dependent conditions, all of which are combined using logical AND or OR
  - Format: `if (` $C_1 \quad \otimes \quad C_2 \quad \otimes \quad \ldots$ `)` $\qquad \otimes \in \{\&\&, ||\}$

Loop-Dependent Conditions

- Linear expressions over index variables i_L of the loops
  - Format: $C_x \simeq \sum_{L=1}^{N} (c_L * i_L) + c \geq 0 \qquad c_L, c \in \mathbb{Z}$

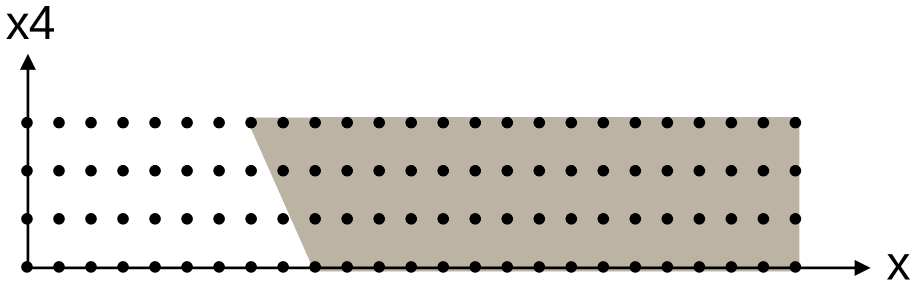
Polytopes & Linear Conditions

Definition (Polyhedra / Polytopes)

- *Polyhedron* $P = \{x \in \mathbb{Z}^N | Ax \geq b\}$ $A \in \mathbb{Z}^{m \times N}, b \in \mathbb{Z}^m$
- Polyhedron P is called *Polytope* iff $|P| < \infty$

Example: Model of Linear Conditions in Nested Loops

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



Convexity of Polytopes

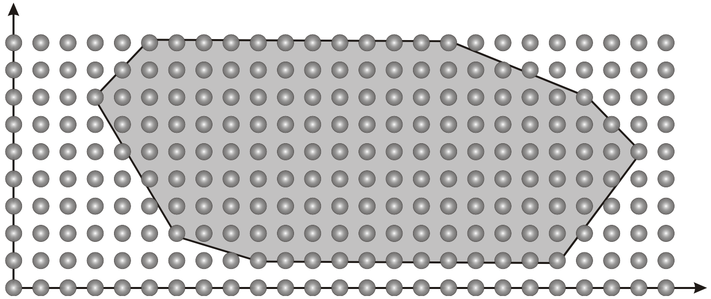
Definition (Convexity)

– A set $S \subset \mathbb{Z}^N$ is called *convex* if each convex combination $ax + by$ of $x, y \in S$ is in S and $(a + b = 1; b \geq 0)$.

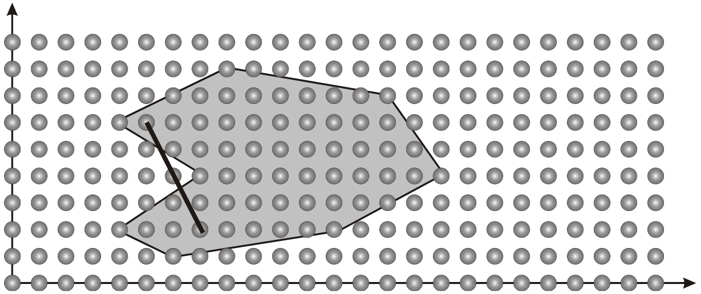
Less Formally

– Each line between any two arbitrary points x and y from S must lie completely in S.

☞ *Polytopes are convex.*



Convex



Not convex

Operations on Polytopes

Properties

- The ***Intersection*** of two polytopes is again a polytope.
- The ***Union*** of two polytopes is not a polytope, since the resulting set is not necessarily convex.

Definition (Finite Union of Polyhedra, FUP)

- For polyhedra $P_1, \ldots, P_n (n \in \mathbb{N}, n < \infty)$, $V := \{P_1 \cup \cdots \cup P_n\}$ is called a *Finite Union of Polyhedra (FUP)*.

Set Operations on FUPs $A = \bigcup_i A_i, B = \bigcup_j B_j$

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

Phase 1 – Condition Satisfiability

Goal

- To identify loop-dependent conditions C_x that constantly evaluate to ‘true’ or ‘false’, irrespectively of the values of index variables of all surrounding loops.

Approach

- Translate each condition C_x into a polytope P_x (☞ *slide 23*)
- Test for the empty set: $P_x == \emptyset \Rightarrow C_x$ always ‘false’
- Test for the universe: $P_x == U \Rightarrow C_x$ always ‘true’

Source Code Modification

- Replacement of all these constantly true or false conditions in the source code by constants ‘0’ or ‘1’.

Phase 2 – Condition Optimization

Given

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

Assumed fictive Situation

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

Assumption: The loops contain only 1 if-statement with only 1 condition. All other if-statements & conditions are quasi suppressed.

Goal: To determine intervals $[l_{C,1}, u_{C,1}] \dots [l_{C,N}, u_{C,N}]$ such that:

- C is satisfied for all loop iterations within these intervals
- Minimization of the number of executions of if-statements after a hypothetical loop nest splitting based on these intervals.

Optimization of a Condition C

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

- Determination of values $l_{C,L}$ and $u_{C,L}$ for every loop L
- Interpretation: C is satisfied for all $l_{C,L} \leq i_L \leq u_{C,L}$
- Optimization goal:

  All values $l_{C,L}$ and $u_{C,L}$ minimize the total amount of executed if-statements

- Simplification: The linearity of C implies

  either $l_{C,L} = l_L$ or $u_{C,L} = u_L$
- Consequence: Determination of only one value $v_{C,L}$

  Either $[v_{C,L}, u_{C,L}]$ or $[l_{C,L}, v_{C,L}]$ satisfies condition C

Some Definitions...

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

- **Total Iteration Space** *(#Executions of the innermost loop's body)*

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

- **Constrained Iteration Space** *(#Executions of the innermost loop's body, constrained to regions specified by values $v_{C,L}$)*

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

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

- **Splitting Loop** *(Index of that loop where splitting will take place)*

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

Illustration (1)

```
Loop 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++)
Loop N=7                 for (y4 = 0; y4 < 4; y4++) {
                           if (4*x+vx+x4-4>35)
                             then_block_1; else else_block_1; }
```

Note:

– Loops are enumerated from 1 ... *N* from the outermost to the innermost loop.

– Only 1 condition in if-statement instead of the many ones from slide 11!

Illustration (2)

```
Loop 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++)
Loop N=7               for (y4 = 0; y4 < 4; y4++) {
                         if (4*x+vx+x4-4>35)
                           then_block_1; else else_block_1; }
```

***TIS*:**

- – Amount of executions of the code within the curly braces.
- – Here: 20 * 36 * 49 * 9 * 9 * 4 * 4 = 45,722,880

Illustration (3)

Loop 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; }
```

Loop

Let $v_{C,1} = 0; v_{C,2} = 10; v_{C,3} = \cdots = 0$ **be given.**

- – Condition C is satisfied for all **x** ≥ 10 and **i**, **y**, ..., **y4** ≥ 0.
- – *CIS*: 20 * (36 – 10) * 49 * 9 * 9 * 4 * 4 = 33,022,080

Illustration (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) ... }
```

CIS: *Number of executions of the code in curly braces for* **x** ≥ 10*.*

Hypothetical Code for $v_{C,2} = 10$

– Splitting Loop $\lambda = 2$ since splitting-if needs to be placed in 2nd loop.

– Question: How often would *all* if-statements shown here be executed?

Counting of If-Statement Executions

- **#If-Statements after Splitting** *(#Original if's + #Splitting-if's)*

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

- **#Original if's** *(Total iterations without constrained iterations)*

  $$IF_{\text{Orig}} = TIS - CIS$$

- **#Splitting-if's**

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

- **#Then-Blocks** *(CIS without iterations of loops inside of λ)*

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

- **#Else-Blocks** *(#Original if's without loop iterations inside of λ)*

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

Illustration (5)

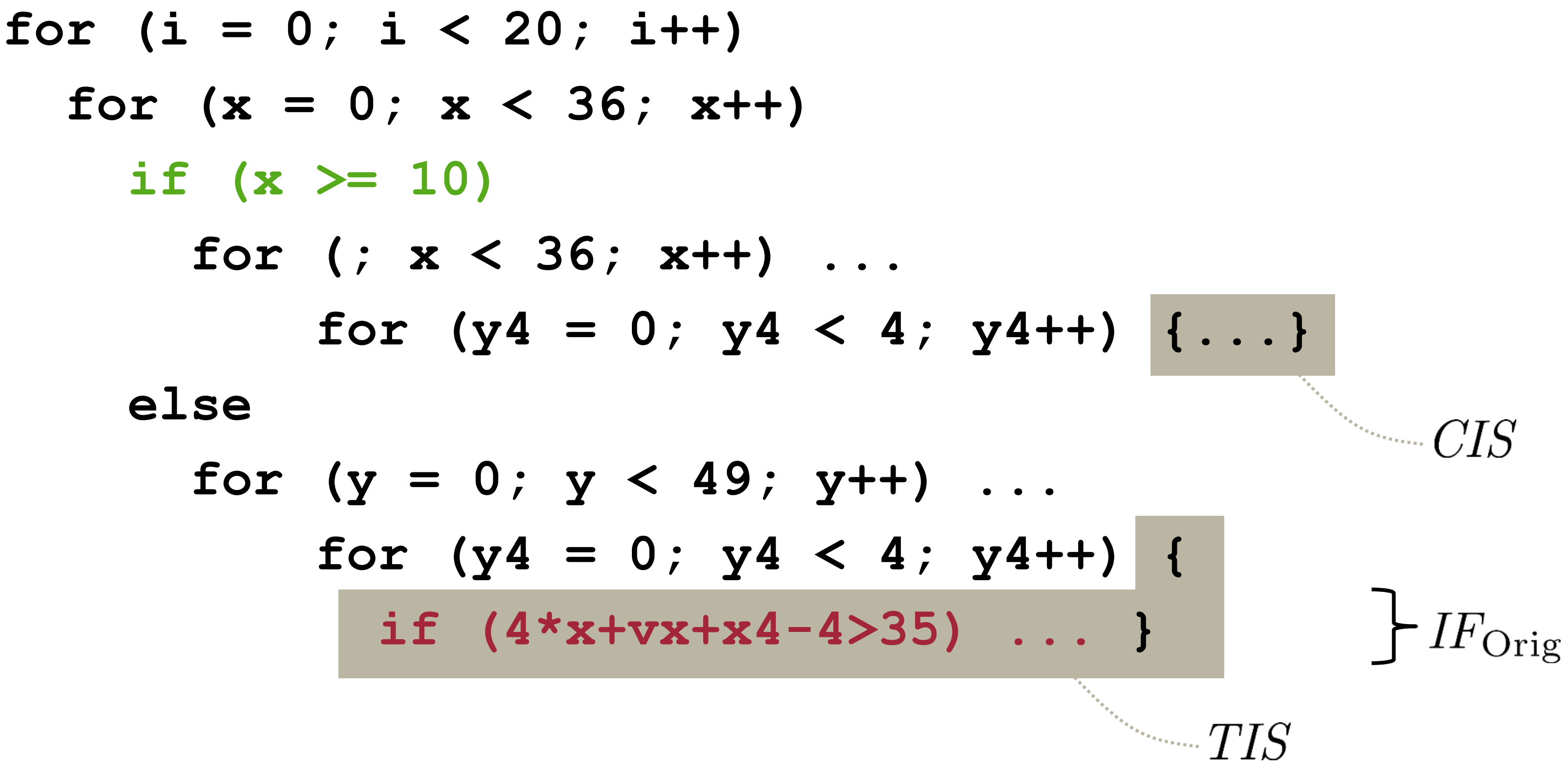
```
for (i = 0; i < 20; i++)
  for (x = 0; x < 36; x++)
    if (x >= 10)
      for (; x < 36; x++) ...
          for (y4 = 0; y4 < 4; y4++) {...}
    else
      for (y = 0; y < 49; y++) ...
          for (y4 = 0; y4 < 4; y4++) {
            if (4*x+vx+x4-4>35) ... }
```

IF_{Split}

IF_{Orig}

IF_{Total}

Illustration (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) ... }
```

$$
\begin{aligned}
IF_{\text{Orig}} &= 20 * 36 * 49 * 9 * 9 * 4 * 4 \; - \\
&\quad 20 * (36 - 10) * 49 * 9 * 9 * 4 * 4 \\
&= 45,722,880 - 33,022,080 = 12,700,800
\end{aligned}
$$

Illustration (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) ... }
```

$\left.\begin{array}{l} \\ \end{array}\right\} IF_{\text{Split}}$

$\#\mathit{Then\text{-}blocks}$

$\#\mathit{Else\text{-}blocks}$

Illustration (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
```

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

```
      for (y = 0; y < 49; y++)
```

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

```
        for (vx = 0; vx < 9; vx++)
```

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

```
          for (vy = 0; vy < 9; vy++)
```

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

```
            for (x4 = 0; x4 < 4; x4++)
```

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

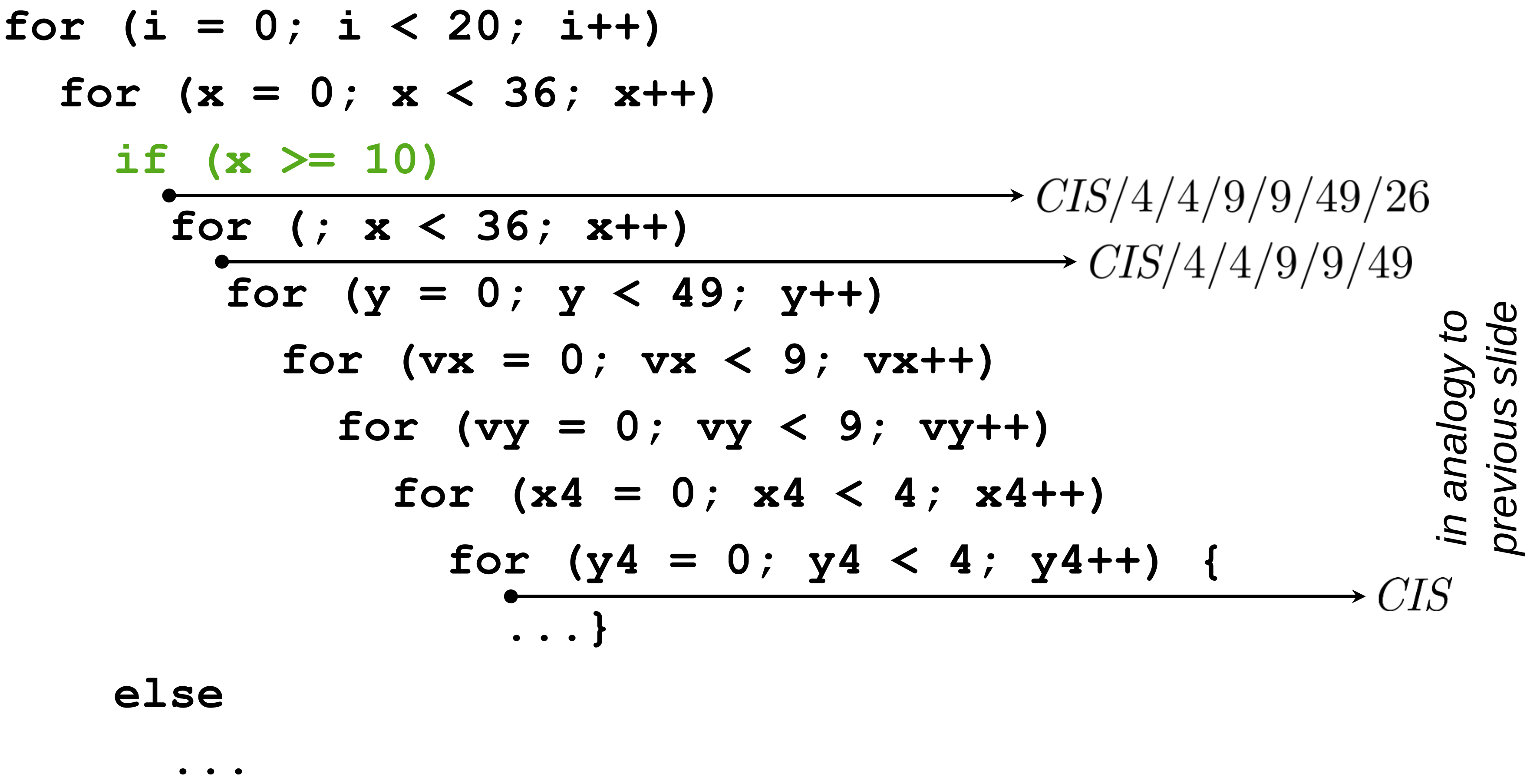
```
              for (y4 = 0; y4 < 4; y4++) {
```

IF_{Orig}

```
                if (4*x+vx+x4-4>35) ... }
```

$$\#\textit{Else-blocks} = IF_{\text{Orig}}/(49 * 9 * 9 * 4 * 4)$$
$$= 12,700,800/63,504 = 200$$

Illustration (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
      ...
```

$CIS/4/4/9/9/49/26$

$CIS/4/4/9/9/49$

CIS

in analogy to previous slide

$$\#\mathit{Then\text{-}blocks} = CIS/(26 * 49 * 9 * 9 * 4 * 4)$$

$$= 33,022,080/1,651,104 = 20$$

Illustration (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) ... }
```

$\}\ IF_{\text{Split}}$

$\}\ IF_{\text{Orig}}$

$\}\ IF_{\text{Total}}$

$$\begin{aligned} IF_{\text{Total}} &= IF_{\text{Orig}} + IF_{\text{Split}} = IF_{\text{Orig}} + \#\textit{Then-Blocks} + \#\textit{Else-Blocks} \\ &= 12,700,800 + 20 + 200 \\ &= 12,701,020 \end{aligned}$$

Computation of Values $v_{C,L}$

Wrap-Up

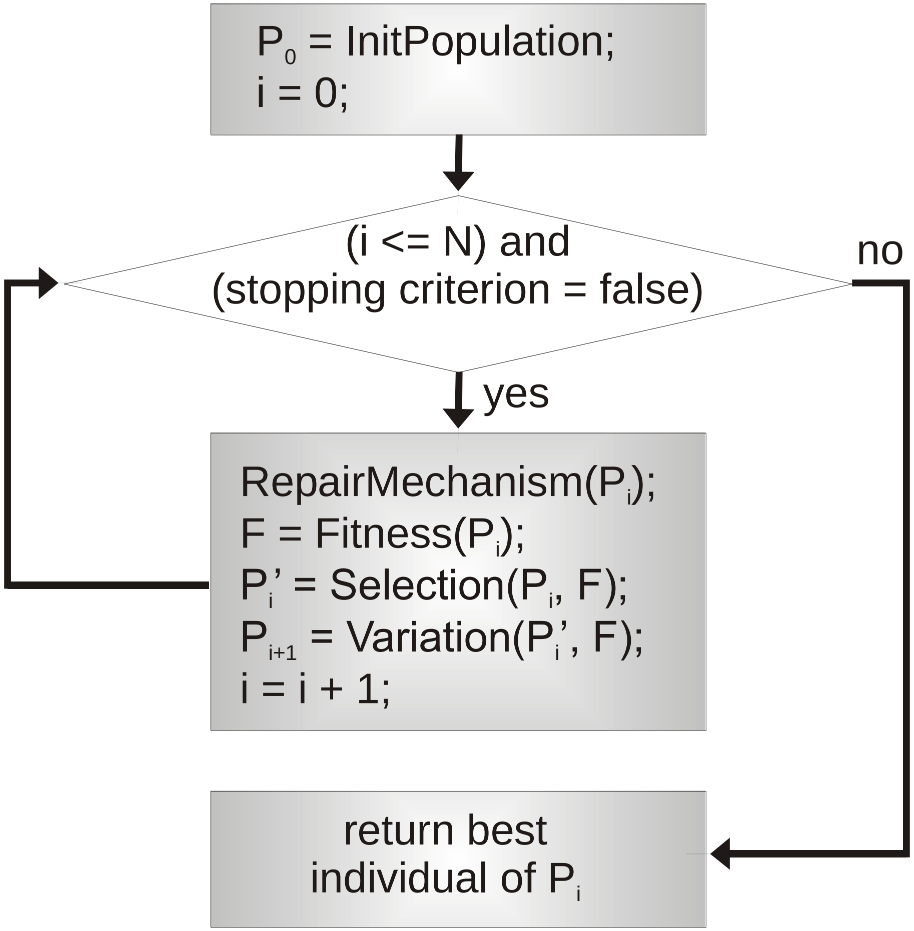
- For a condition C and given values $v_{C,L}$, we can compute how many if-statements would be executed after splitting based on $v_{C,L}$.

Very nice, but...

... who produces good values for $v_{C,L}$?

☞ **A Genetic Algorithm**

Workflow of Genetic Algorithms (1)



P_0 = InitPopulation;
i = 0;

(i <= N) and
(stopping criterion = false)

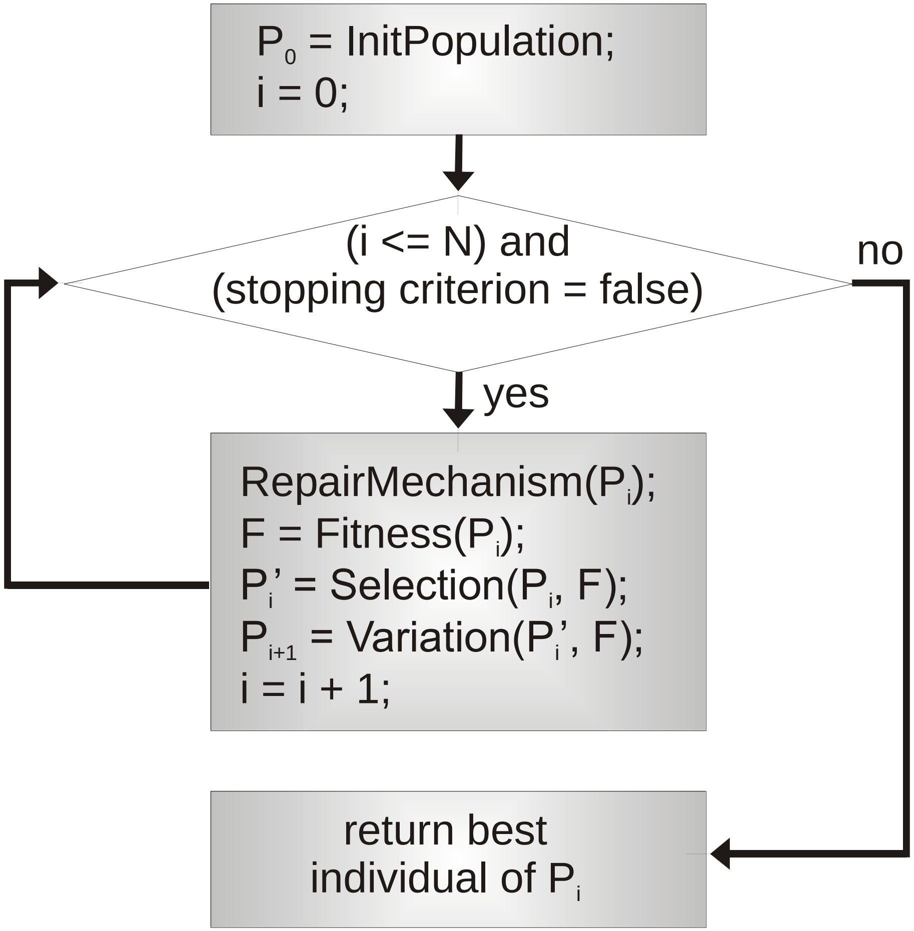
RepairMechanism(P_i);
F = Fitness(P_i);
P_i' = Selection(P_i, F);
P_{i+1} = Variation(P_i', F);
i = i + 1;

return best individual of P_i

- In the style of natural evolution, *“survival of the fittest”*
- Optimization loop *i* = 0, 1, ...
- Each iteration *i* maintains *population* P_i; a population contains several *individuals*
- An individual represents one possible solution for the modeled optimization problem

<INTERLUDE>

Workflow of Genetic Algorithms (2)



P_0 = InitPopulation;
i = 0;

(i <= N) and
(stopping criterion = false)

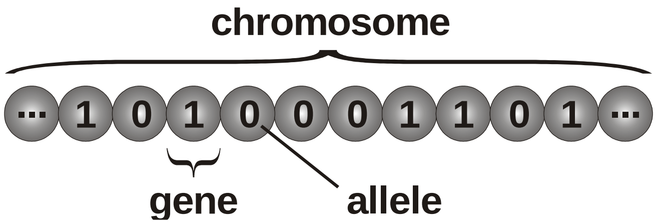
yes

no

RepairMechanism(P_i);
F = Fitness(P_i);
P_i' = Selection(P_i, F);
P_{i+1} = Variation(P_i', F);
i = i + 1;

return best
individual of P_i

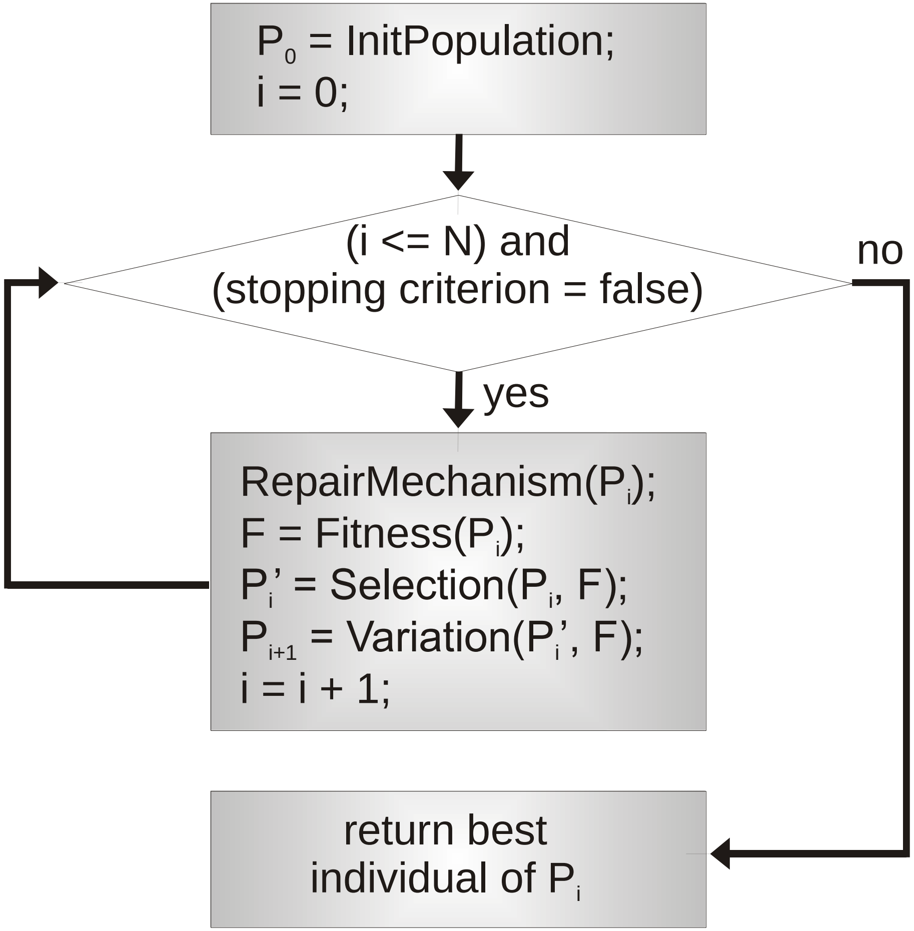
- An individual's data structure is called *chromosome*.



- A chromosome consists of many *genes* that are used to save data.
- One actual value stored in a gene is called *allele*.

<INTERLUDE>

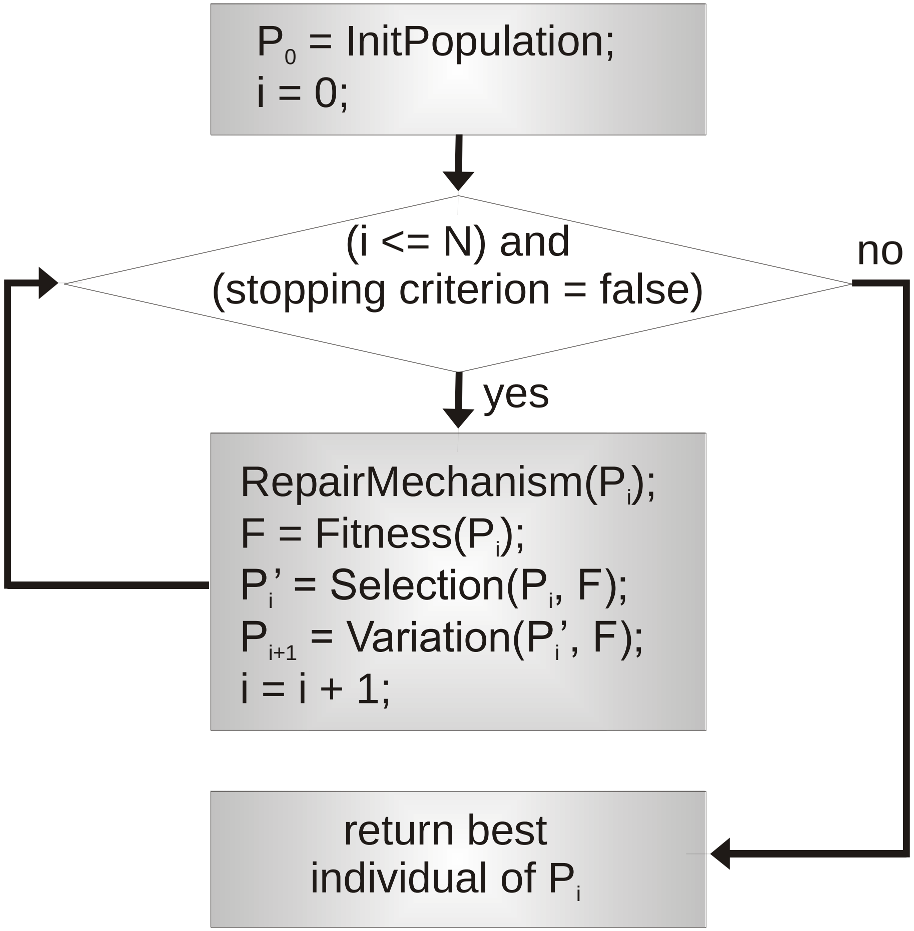
Workflow of Genetic Algorithms (3)



- A fitness function computes the *fitness* of each individual inside P_i.
- From P_i, a subset P_i' of highest / lowest fitness is selected (*selection*, depending on whether a minimization or maximization problem is optimized).
- P_i' is completed to the next population P_{i+1} by randomly generating new individuals *(variation)*

<INTERLUDE>

Workflow of Genetic Algorithms (4)



P_0 = InitPopulation;
i = 0;

(i <= N) and
(stopping criterion = false)

no

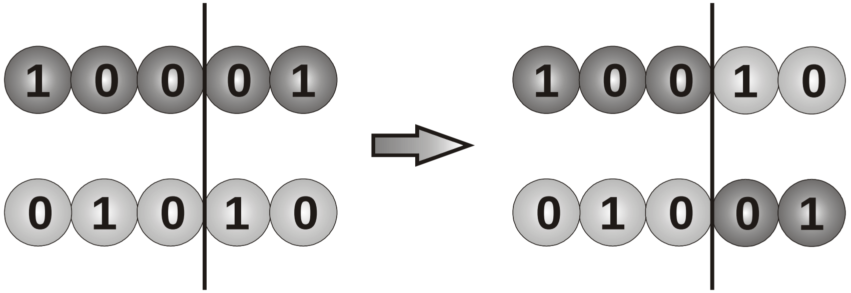
yes

RepairMechanism(P_i);
F = Fitness(P_i);
P_i’ = Selection(P_i, F);
P_{i+1} = Variation(P_i’, F);
i = i + 1;

return best
individual of P_i

– Variation makes use of two basic genetic operators:

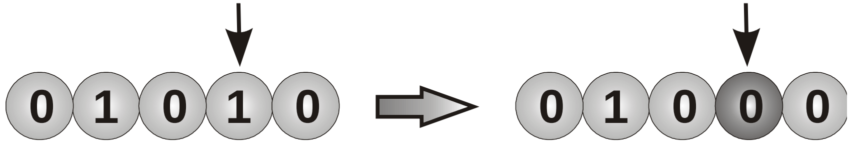
– *Cross-over:*



1 0 0 0 1 → 1 0 0 1 0

0 1 0 1 0 → 0 1 0 0 1

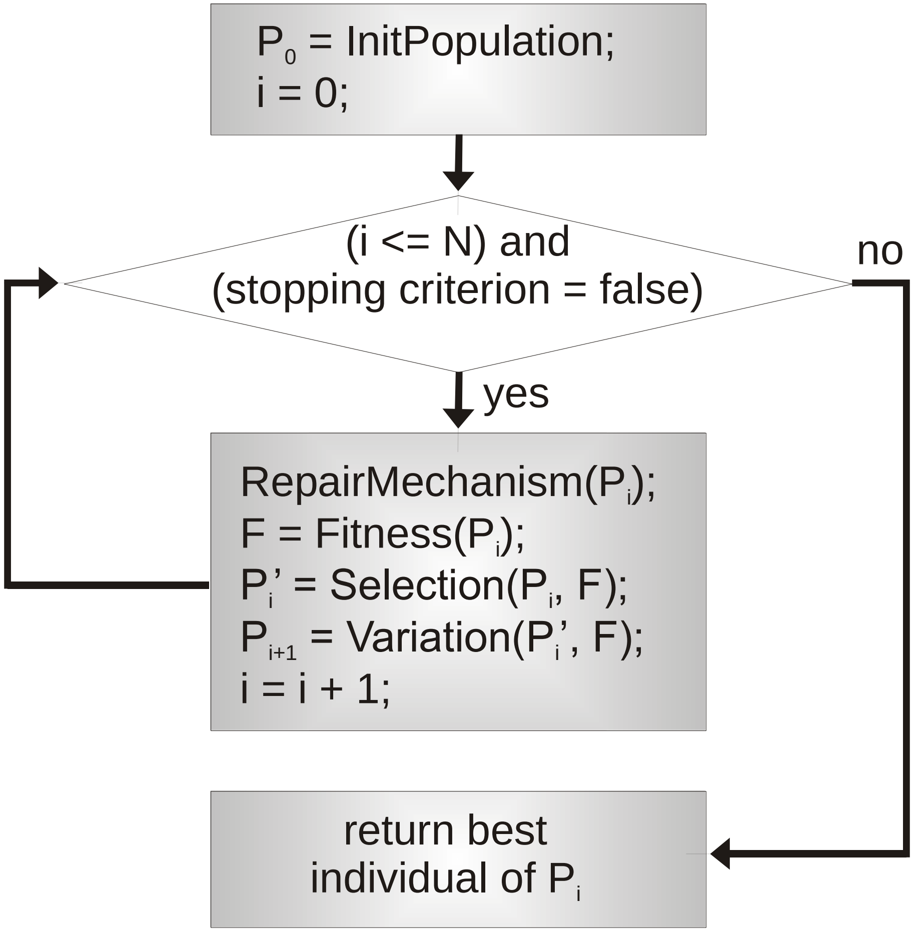
– *Mutation:*



0 1 0 1 0 → 0 1 0 0 0

<INTERLUDE>

Workflow of Genetic Algorithms (5)



P_0 = InitPopulation;
i = 0;

(i <= N) and
(stopping criterion = false)

yes

no

RepairMechanism(P_i);
F = Fitness(P_i);
P_i' = Selection(P_i, F);
P_{i+1} = Variation(P_i', F);
i = i + 1;

return best
individual of P_i

- Due to the randomness in variation, P_i can contain individuals that do not represent valid solutions: *Repair* mechanism.
- Termination of the GA if
  - max. *N* iterations reached,
  - best observed fitness unchanged for *y* iterations,
  - ...
- Final result is that individual from last population with best fitness.

</INTERLUDE>

Genetic Algorithm for Condition Optimization

Chromosomal Representation

- For *N* nested loops, each chromosome has *N* genes
- Each gene holds one integer number
- Gene *L* represents the value $v_{C,L}$ to be optimized
- Domain of each gene *L* restricted to interval $[l_L, u_L]$

Fitness

- Fitness of an individual = IF_{Total}
- Invalid individuals $(v_{C,1}, \ldots, v_{C,N})$ represent iterations within the loop nest in which condition C is not always satisfied
- Invalid individuals obtain a very poor fitness

Result of Condition Optimization

Inputs to Condition Optimization

- Linear condition C
- Loop bounds $[l_L, u_L]$

Output of the Genetic Algorithm

- Values $(v_{C,1}, \ldots, v_{C,N})$ of the individual with best fitness

Output of Condition Optimization Phase

- Polytope

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

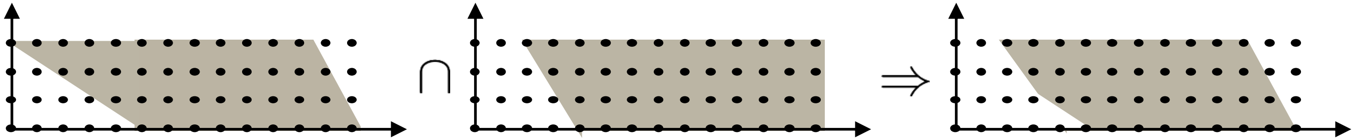
Phase 3 – Search Space Generation

Given: If-statements, conditions & polytopes

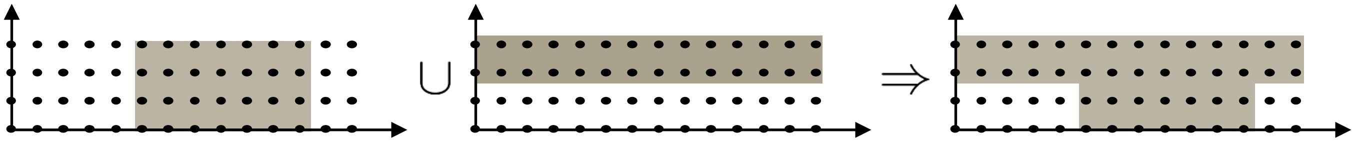
$$\mathrm{IF}_i = (C_{i,1} \otimes C_{i,2} \otimes \cdots \otimes C_{i,n}), \otimes \in \{\&\&, \,|\,|\} \qquad \forall C_{i,j} \rightsquigarrow P'_{i,j}$$

Construction of FUPs *P*$_i$ for each full if-statement IF$_i$

- $\text{if } C_{i,j-1} \;\; \&\& \;\; C_{i,j} :$



- $\text{if } C_{i,j-1} \;\; |\,| \;\; C_{i,j} :$



Construction of a Global FUP *(Global Search Space)*

FUP *G* models iteration space in which *all* if-statements are satisfied.
Constructed by intersecting all FUPs *P*$_i$ of the individual if-statements:

- $G = \bigcap P_i$

Phase 3 – Structure of the FUP *G*

Consequence of Using the $\cup$ Operator on the Previous Slide

- *G* is a finite union of polytopes and can thus be seen as:
- $G = R_1 \cup R_2 \cup \cdots \cup R_M$
- Interpretation:

  Each polytope R_r denotes

  *one region in the iteration space of the entire loop nest in which **all** if-statements are satisfied.*

Global Search Space for MPEG Code

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

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

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

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

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

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

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

- $G = P_1 \cap P_2 =$

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

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

Global Search Space & Splitting-If (1)

- $G = \mathrm{IF}_1 \cap \mathrm{IF}_2 =$

$$\{\mathrm{x} = 0 \ \wedge \ \mathrm{vx} = 0 \ \wedge \ \mathrm{y} \geq 13\} \ \cup \ \{\mathrm{x} \geq 10\} \cup$$
$$\{\mathrm{y} = 0 \ \wedge \ \mathrm{vy} = 0 \ \wedge \ \mathrm{x} \geq 9\} \ \cup \ \{\mathrm{y} \geq 14\}$$

Direct Translation of *G* into Splitting-If

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

Not a Good Idea

☞ This splitting-if must be placed in **`vy`**-loop (3rd-innermost!)

☞ Leads to 10,103,760 executions of if-statements in total

Global Search Space & Splitting-If (2)

- $G = \mathrm{IF}_1 \cap \mathrm{IF}_2 =$

$$\{\mathrm{x} = 0 \ \wedge \ \mathrm{vx} = 0 \ \wedge \ \mathrm{y} \geq 13\} \ \cup \ \{\mathrm{x} \geq 10\} \cup$$
$$\{\mathrm{y} = 0 \ \wedge \ \mathrm{vy} = 0 \ \wedge \ \mathrm{x} \geq 9\} \ \cup \ \{\mathrm{y} \geq 14\}$$

Alternative: Use only sub-polytopes R_r of *G* for splitting-if.
Legal, since each sub-polytope R_r by itself already satisfies all if-statements.

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

- This sub-polytope is also not a good solution
  - ☞Leads to 25,401,820 if-statement executions

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

- This combination of sub-polytopes is a good solution
  - ☞Leads to 7,261,120 if-statement executions

Phase 4 – Search Space Exploration

GA: Selects regions from $G = R_1 \cup R_2 \cup \dots \cup R_M$

- Individual: Bit-vector that marks the selected regions

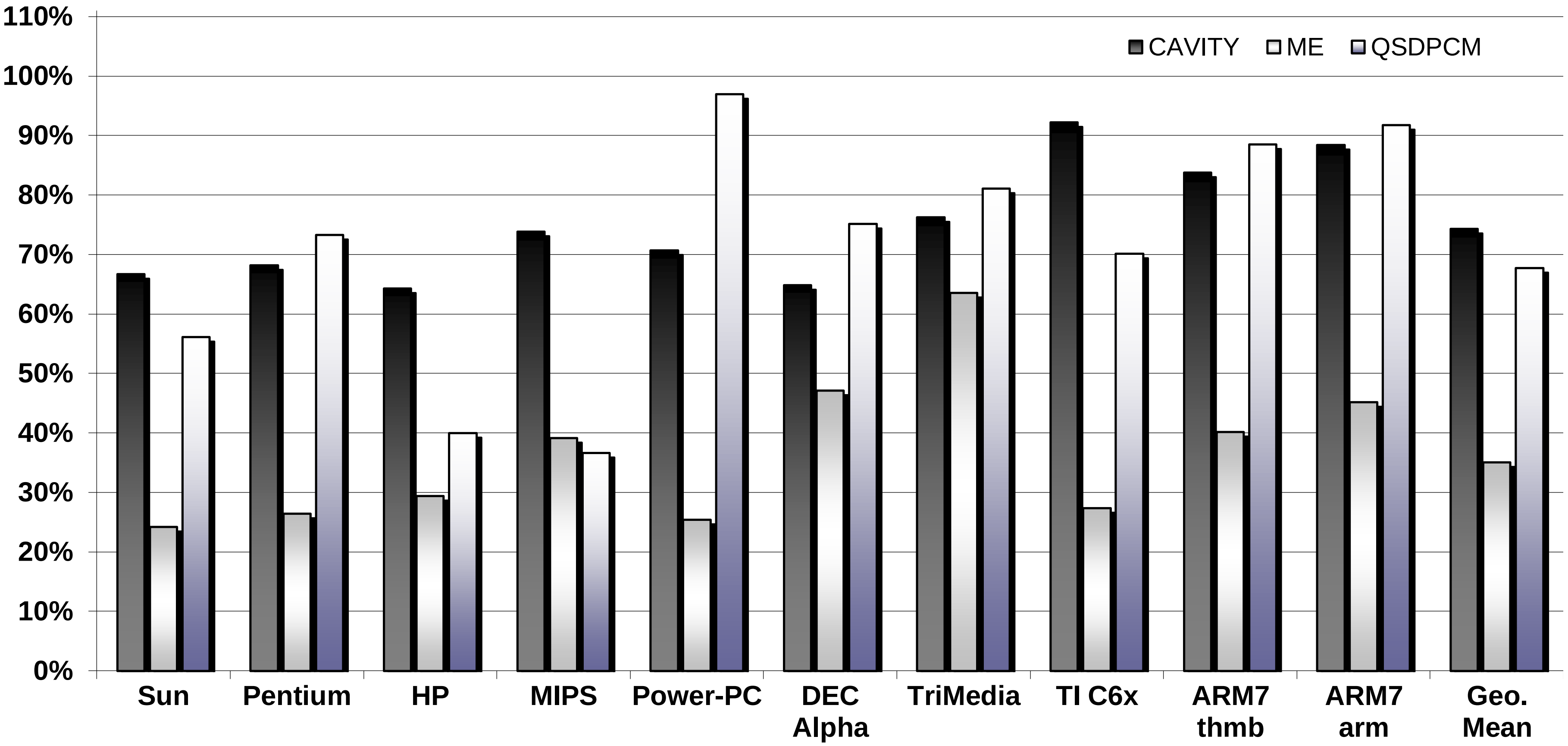
  Individual $I = (I_1, \dots, I_M), I_r = \begin{cases} 1 & \text{if region } R_r \text{ selected,} \\ 0 & \text{otherwise} \end{cases}$

- Fitness function computes again #{If-statement executions} after loop nest splitting
- Fitness function minimized by GA

Resulting Splitting-If

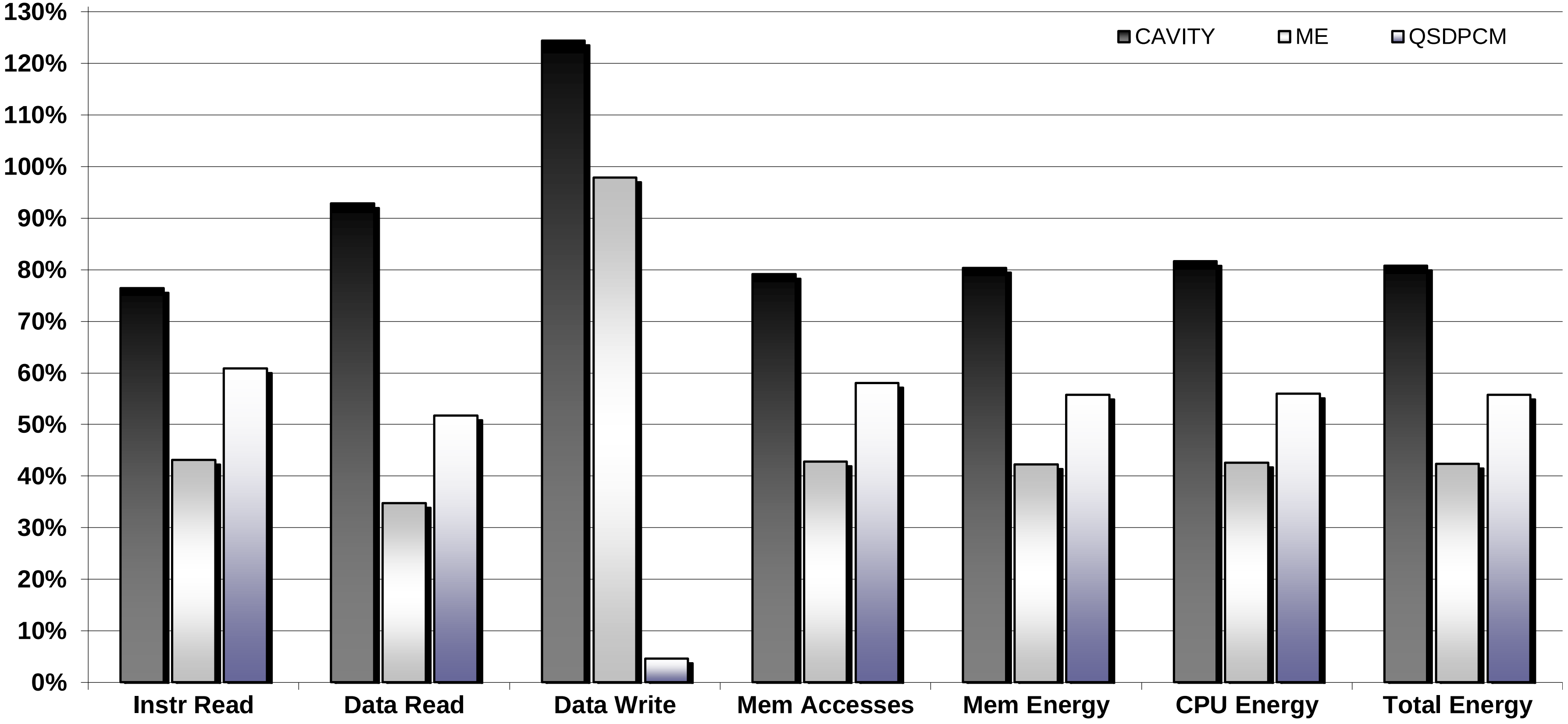
- Placed in the outermost possible loop of the loop nest
- Contains all conditions and operators as specified by the selected regions R_r

Relative Run-Times after Loop Nest Splitting



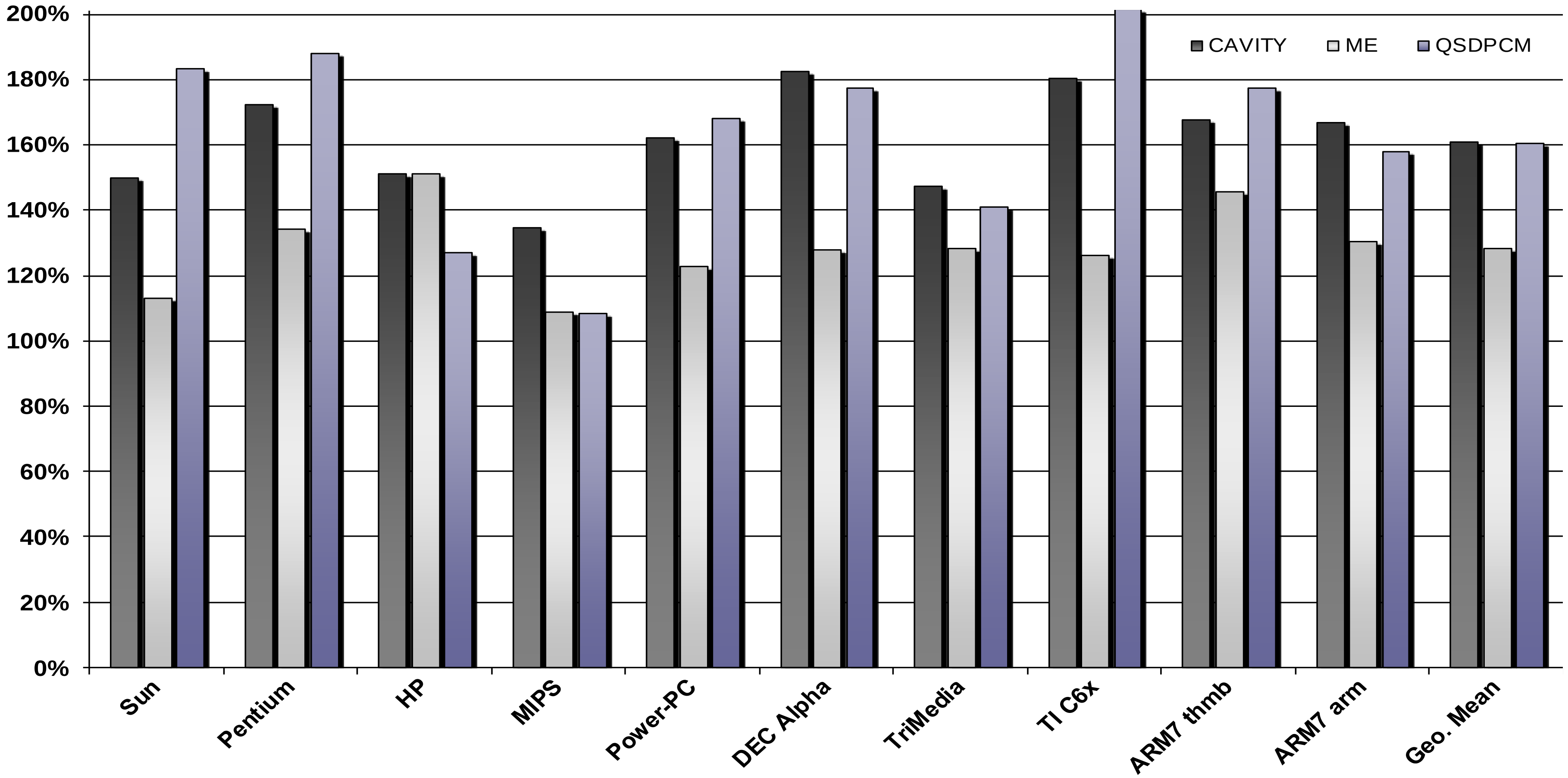
100% = Run-times of the benchmarks without Loop Nest Splitting

Relative Energy Consumption (ARM7) after LNS



100% = Values of the Benchmarks without Loop Nest Splitting

Relative Code Size after Loop Nest Splitting



100% = Size of the benchmarks without Loop Nest Splitting

References

Loop Nest Splitting

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

– H. Falk. *Control Flow Optimization by Loop Nest Splitting at the Source Code Level*. University of Dortmund, Technical Report No. 773, Dortmund 2003.

Summary

Non-Compiler Optimizations

- *Post-pass* after compiler, e.g., at linker-level
- *Pre-pass* before compiler, at source code level

Loop Nest Splitting

- Control flow optimization in data flow-dominated multimedia applications
- Polytopes used to model linear conditions and loops
- Genetic algorithms used to optimize polytope models
- Significant reductions in terms of ACET and energy *(and WCET)*, but partially heavy code size increases