CHAIR FOR COMPILER CONSTRUCTION

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

RoSI Lecture: Optimizing Compilers and Role-based Programming

Jeronimo Castrillon
TU Dresden – Cfaed
WS 2016

Chair for Compiler Construction
Georg-Schumann-Str. 7A, 2 OG
jeronimo.castrillon@tu-dresden.de

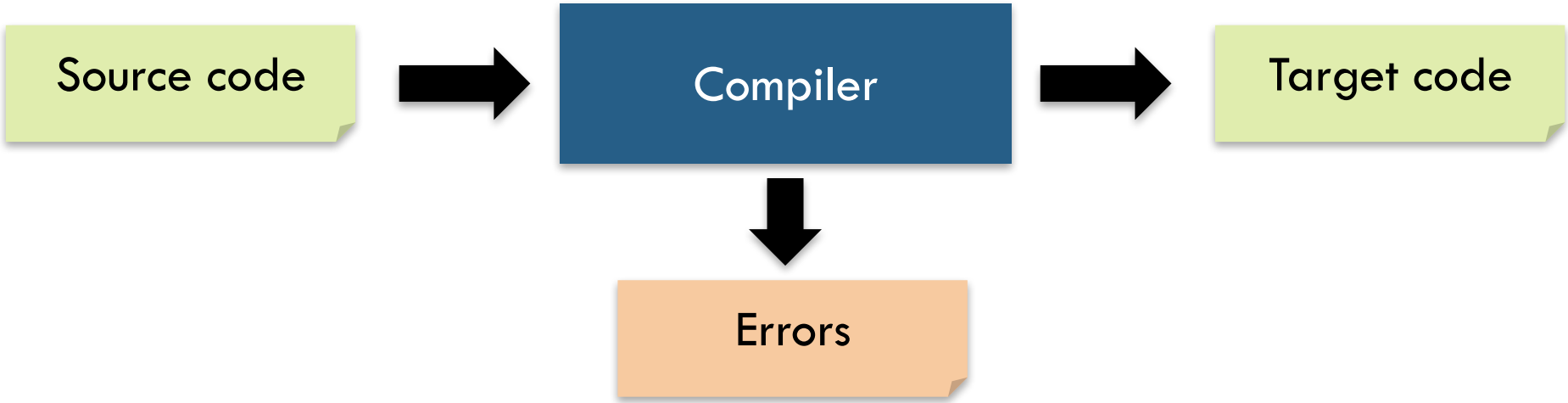
Agenda

- Introduction
- Sample optimizations
- Opportunities in role-based programming

Introduction

- Terminology
- Abstract view
- Optimizing compilers

Compilation



- Compiler translates an input source code to a target code
- Typical: target code closer to machine code (e.g., C ➔ assembly)
- Must recognize illegal code and generate correct code
- Must agree with lower layers (e.g., storage, linker and runtime)

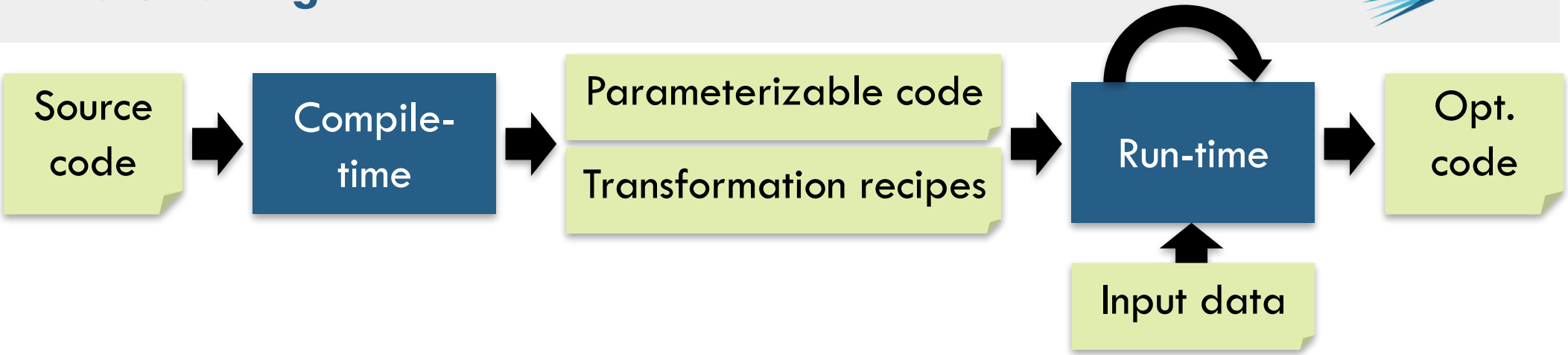
Interpretation

- Interpreter: Does not generate another program – stepwise translation
  ➔ "Easier" to implement but less room for optimization
- Translation either while parsing or from an intermediate code
- Examples: scripting languages (e.g., Perl, Matlab), Java byte-code is interpreted by the JVM (nowadays: just-in-time (JIT) compiled)

Just-in-time compilation

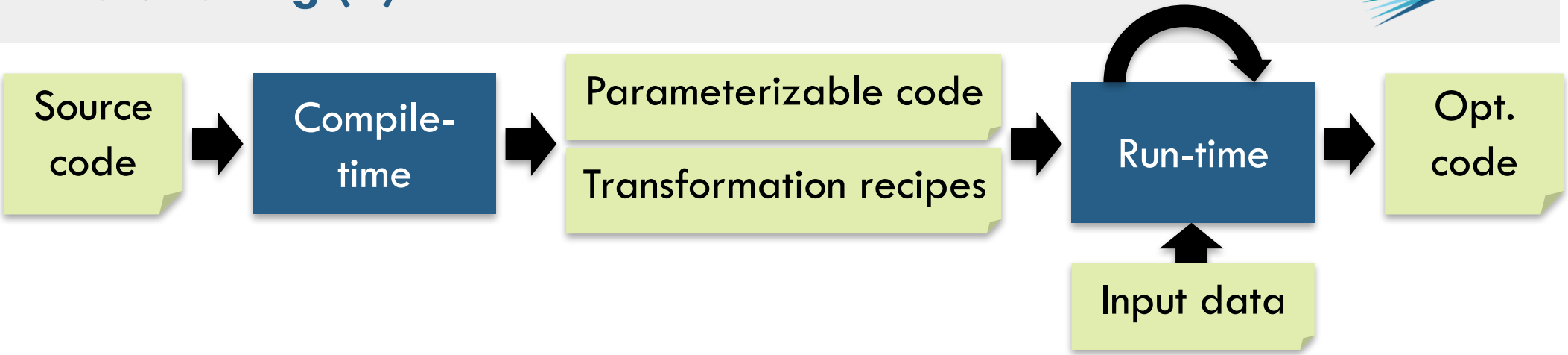
- Combines performance of compilation with flexibility of interpretation
- **Dynamically translate** intermediate code into target code
- Continuously analyze the running application
  - Identify portions of code it is worth compiling instead of interpreting
- Common for dynamic languages
- Related to dynamic binary translation (processor simulation/emulation)

Auto-tuning



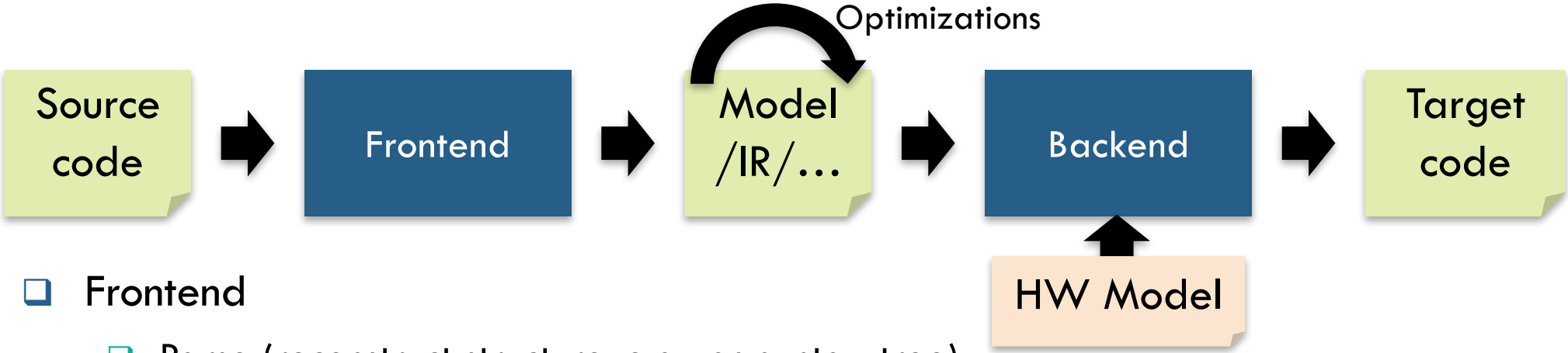
- Generate a searchable space (a way of generating code variants)
  - The space may be controlled by parameters (e.g., size of matrixes)
- Perform design-space exploration (DSE) at run-time to find the best variant for a given set of inputs (and parameters)
- The why: Architectures becoming too complex → limits of static (& data-independent) optimizations

Auto-tuning (2)



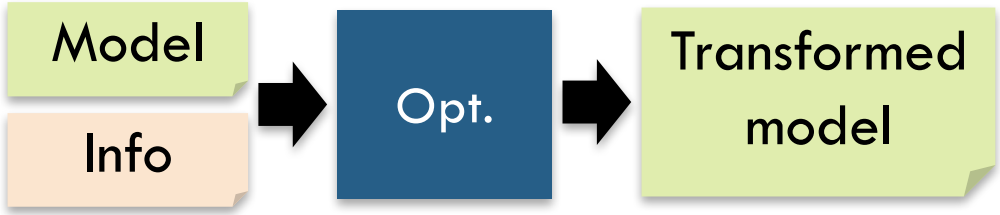
- Can be
  - Library-based: search implemented within the function call (e.g., FFTW)
  - Application-specific with language support to specify parameters (Petabricks)
  - Compiler-based parameter search, e.g., unroll factor, data layout (Spiral, CHiLL)

Abstract view of a compiler



- Frontend
  - Parse (reconstruct structure, e.g., as syntax tree)
  - Semantic checks: types, scoping, …
- Model/Intermediate representation (IR)
  - To some extent architecture agnostic
  - Comes at different levels of abstraction
- Backend: Pattern matching, rewriting and architecture-dependent optimizations

Optimizing compilers



- All of them are by now optimizing
- Optimization goals
  - Correctness (semantics preserving)
  - Multi-objectives: performance, code size, energy, power, robustness, …
- Optimization requires extracting/collecting information
  - Control and dataflow analysis
  - Profiling information (execution paths, counts, …)
  - Memory disambiguation
  - …

More terminology

- Control flow analysis: Study branching and paths in the program
- Dataflow analysis: Study definitions, uses, liveness of program objects
- Context sensitive analysis: Analysis aware of the calling context of a function
- Local/global/inter-procedural analysis: within a basic-block/function/program

Agenda

- Introduction
- **Sample optimizations**
- Opportunities in role-based programming

Sample optimizations

- LLVM IR
- Function inlining
- Parallelizing compilers
- Dataflow programs

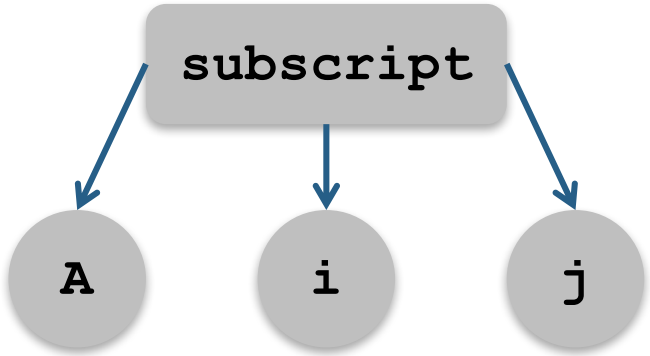
Intermediate representation

- Different levels of IR depending on what you want to analyze/optimize
- Example: Accessing a 2D array with 10 columns (row-oriented)

A[i][j]

```
loadI   1,  R1
sub     Rj, R1, R2 // R2 = Rj-R1
loadI   10, R3
mult    R2, R3, R4 // R4 = R2*R3
sub     Ri, R1, R5
add     R4, R5, R6
loadI   @A, R7
add     R7, R6, R8
load    R8, RAij
```

Famous 3-address-code



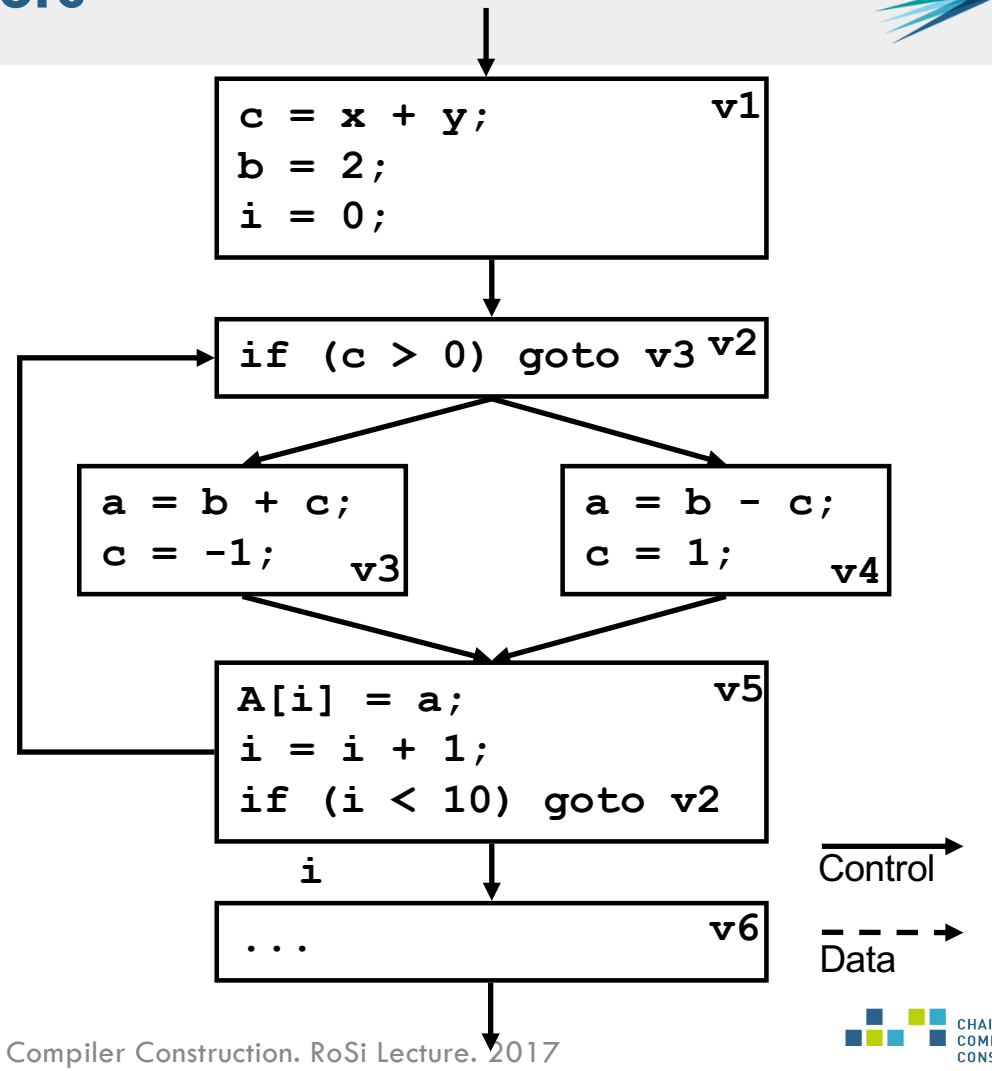
Higher-level representation (e.g., for memory disambiguation)

Adapted from: http://www.inf.ed.ac.uk/teaching/courses/ct/slides/Lecture09.pdf

IR and analysis – Reaching defs

```
1:   ...
2:   c = x + y;
3:   b = 2;
4:   for (i = 0; i < 10; i++) {
5:     if (c > 0) {
6:       a = b + c;
7:       c = -1;
8:     } else {
9:       a = b - c;
10:      c = 1;
11:    }
12:    A[i] = a;
13:  }
14:  ...
```

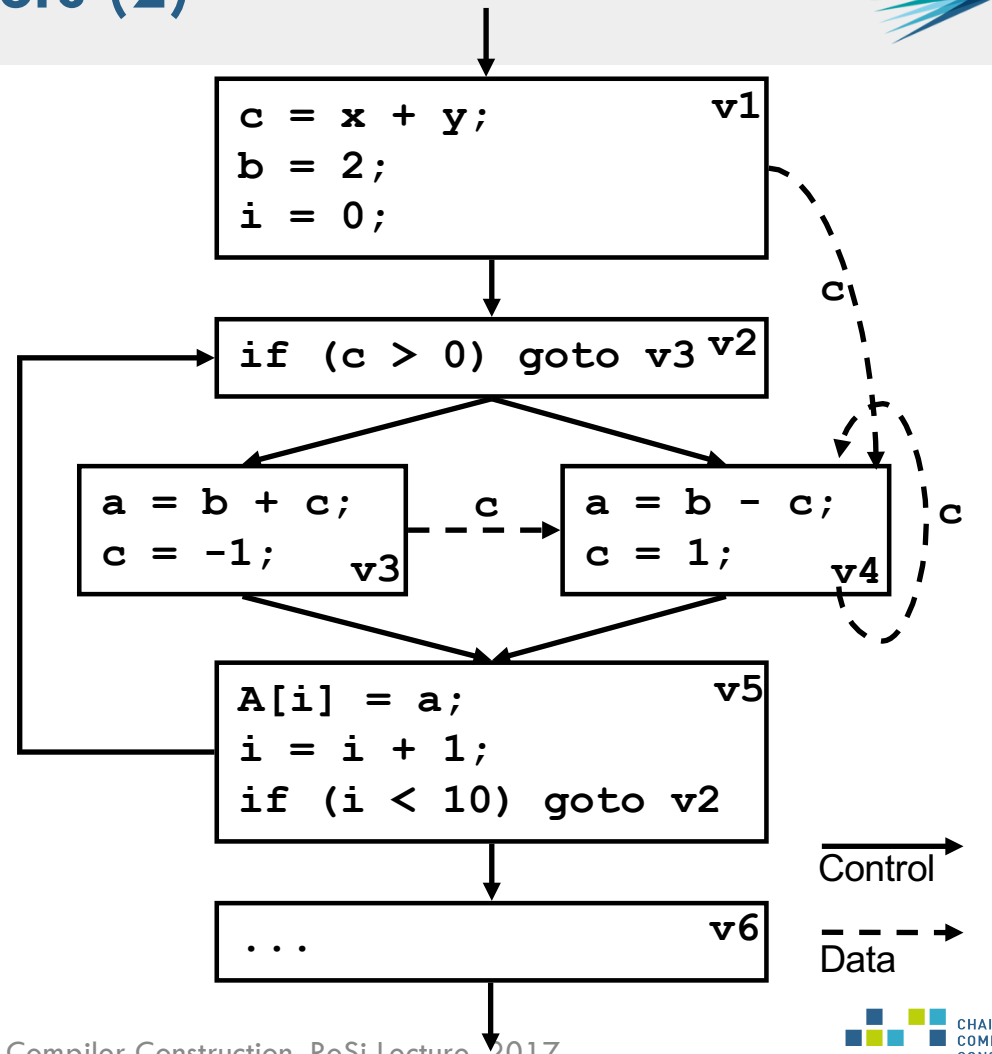
Which definitions of c may reach this statement?



IR and analysis – Reaching defs (2)

```
1:   ...
2:   c = x + y;
3:   b = 2;
4:   for (i = 0; i < 10; i++) {
5:     if (c > 0) {
6:       a = b + c;
7:       c = -1;
8:     } else {
9:       a = b - c;
10:      c = 1;
11:    }
12:    A[i] = a;
13:  }
14:  ...
```

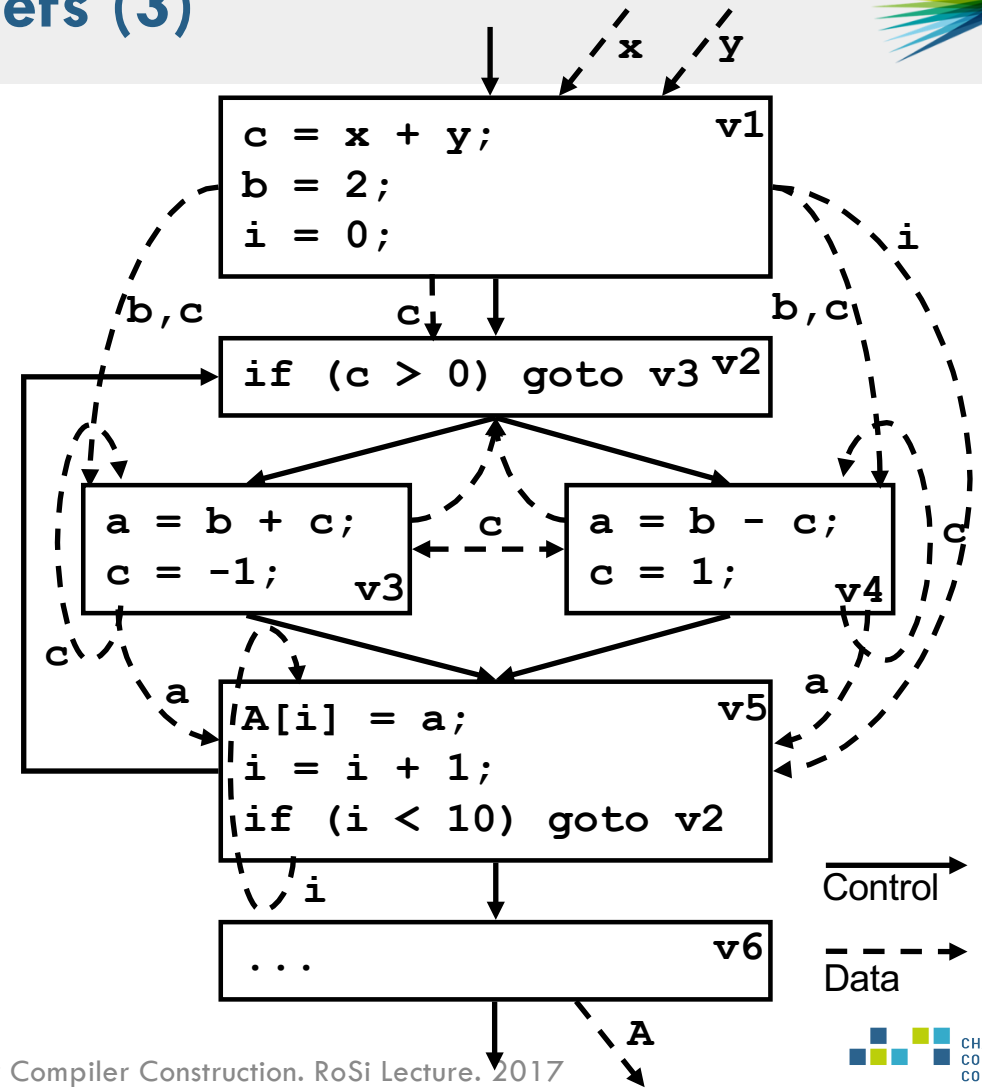
Which definitions of c may reach this statement?



IR and analysis – Reaching defs (3)

```
1:   ...
2:   c = x + y;
3:   b = 2;
4:   for (i = 0; i < 10; i++) {
5:     if (c > 0) {
6:       a = b + c;
7:       c = -1;
8:     } else {
9:       a = b - c;
10:      c = 1;
11:    }
12:    A[i] = a;
13:  }
14:  ...
```

Which definitions of c may reach this statement?



IR and analysis – Reaching defs (4)

- Potential for optimization

```
...
c = x + y;
b = 2;
for (i = 0; i < 10; i++) {
  if (c > 0) {
    a = b + c;
    c = -1;
  } else {
    a = b - c;
    c = 1;
  }
  A[i] = a;
}
...
```

```
...
c = x + y;
c = c > 0 ? c: -c;
A[0] = 2 + c;
for (i = 1; i < 10; i++)
  A[i] = 3;
...
```

If A had100.000 elements – Speedup of 10x

Compiler optimization gets close to this

Example on LLVM IR

- Command: clang –S <opt. flag> -emit-llvm <source> -o <target>

```
2:   c = x + y;
3:   b = 2;
4:   for (i = 0; i < 10; i++) {
5:     if (c > 0) {
6:       a = b + c;
7:       c = -1;
8:     } else {
9:       a = b - c;
10:      c = 1;
11:    }
12:    A[i] = a;
13:  }
```

```
58  ; <label>:4
59    %5 = load i32, i32* %i, align 4
60    %6 = icmp slt i32 %5, 100000
61    br i1 %6, label %7, label %26
62
63  ; <label>:7
64    %8 = load i32, i32* %c, align 4
65    %9 = icmp sgt i32 %8, 0
66    br i1 %9, label %10, label %14
67
68  ; <label>:10
69    %11 = load i32, i32* %b, align 4
70    %12 = load i32, i32* %c, align 4
71    %13 = add nsw i32 %11, %12
72    store i32 %13, i32* %a, align 4
73    store i32 -1, i32* %c, align 4
74    br label %18
75
76  ; <label>:14
77    %15 = load i32, i32* %b, align 4
78    %16 = load i32, i32* %c, align 4
79    %17 = sub nsw i32 %15, %16
80    store i32 %17, i32* %a, align 4
81    store i32 1, i32* %c, align 4
82    br label %18
83
84  ; <label>:18
85    %19 = load i32, i32* %a, align 4
86    %20 = load i32, i32* %i, align 4
87    %21 = sext i32 %20 to i64
88    %22 = getelementptr inbounds [100000 x i32], [100000 x i32]* @A, i64 0, i64 %21
89    store i32 %19, i32* %22, align 4
90    br label %23
91
92  ; <label>:23                                    ; preds = %18
93    %24 = load i32, i32* %i, align 4
94    %25 = add nsw i32 %24, 1
95    store i32 %25, i32* %i, align 4
96    br label %4
97
98  ; <label>:26
99    ret i32 0
100 }
```

Loop condition

If condition

True branch (add)

False branch (sub)

Array access

Un-optmized LLVM IR

Example on LLVM IR (2)

- Command: clang –S <opt. flag> -emit-llvm <source> -o <target>

```
2:   c = x + y;
3:   b = 2;
4:   for (i = 0; i < 10; i++) {
5:     if (c > 0) {
6:       a = b + c;
7:       c = -1;
8:     } else {
9:       a = b - c;
10:      c = 1;
11:    }
12:    A[i] = a;
13:  }
```

```
; <label>:4
  %5 = load i32, i32* %i, align 4
  %6 = icmp slt i32 %5, 100000
  br i1 %6, label %7, label %26

; <label>:7
  %8 = load
  %9 = icmp
  br i1 %9,

; <label>:1
  %11 = lo
  %12 = lo
  %13 = add
  store i3
  store i3
  br label

; <label>:1
  %15 = lo
  %16 = lo
  %17 = sub
  store i3
  store i3
  br label

; <label>:1
  %19 = lo
  %20 = lo
  %21 = sext i32 %20 to i64
  %22 = getelementptr inbounds
  store i32 %19, i32* %22, alig
  br label %23

; <label>:23                                  ; preds = %18
  %24 = load i32, i32* %i, align 4
  %25 = add nsw i32 %24, 1
  store i32 %25, i32* %i, align 4
  br label %4

; <label>:26
  ret i32 0
}
```

Loop condition

```
; <label>:1
  %indvars.iv = phi i64 [ 0, %0 ], [ %indvars.iv
  %c.01 = phi i32 [ 5, %0 ], [ %c.1.1, %1 ]
  %2 = icmp sgt i32 %c.01, 0
  %3 = sub nsw i32 0, %c.01
  %a.0.p = select i1 %2, i32 %c.01, i32 %3
  %a.0 = add nsw i32 %a.0.p, 2
  %4 = getelementptr inbounds [100000 x i32], [100000          * @A, i64 0, i64 %indvars.iv
  store i32 %a.0, i32* %4, align 8, !tbaa !2
  %indvars.iv.next = or i64 %indvars.iv, 1
  %c.1.1 = select i1 %2, i32 1, i32 -1
  %5 = getelementptr inbounds [100000 x i32], [100000 x i32]* @A, i64 0, i64 %indvars.iv.next
  store i32 3, i32* %5, align 4, !tbaa !2
  %indvars.iv.next.1 = add nsw i64 %indvars.iv, 2
  %exitcond.1 = icmp eq i64 %indvars.iv.next.1, 100000
  br i1 %exitcond.1, label %6, label %1

; <label>:6
  ret i32 0
}
```

Store condition (c>0)

Use condition in select

Two array stores

Loop condition

Optmized LLVM IR (O3)

A[i] = 3 (half of the times)!

Un-optmized LLVM IR

Optimizations

- So far
  - Graph representation of program
  - Sample analysis (reaching definitions) and what can be achieved

- Sample optimizations: Focus on required information, problem formulation and solution approach
  - Function inlining: Inter-procedural analysis, profiling information
  - Parallelization: Detailed dependence analysis (restricted problem formulation)
  - Dataflow programs: Move to parallel semantics, time-annotations, static schedules

Function inlining

- Inlining: Replace function call by inserting the function body in the code
- Why
  - Increase potential for other optimizations (w/o complex inter-procedural analysis)
  - Reduce stack management overhead
  - Remove jumps (call and return)
- Common example for code-size performance tradeoff

- Example

```
int f1(int x)
{ return x+1; }

int f2(int x)
{ return f1(x)+2; }
```

➡

```
int f2(int x)
{ return x+1+2; }
```

➡

```
int f2(int x)
{ return x+3; }
```

Function inlining (2)

- Lots of support in compilers
- Intuition
  - Static calls impact in code size
  - Dynamic calls impact performance (need profiling)
  - Inline
    - Small functions (comparable to calling overhead)
    - Single static call
    - Static calls within loops (high dyn. calls)
    - Functions with single switch-case and often called with constant parameters

```
-fno-inline
-finline-small-functions
-findirect-inlining
-finline-functions
-finline-functions-called-once
-fearly-inlining
-finline-limit=n
-fno-keep-inline-dllexport
-fkeep-inline-functions
-fpartial-inlining
-flto[=n] (for linked-time opt)
--param name=value
  max-inline-insns-single
  max-inline-insns-auto
  inline-min-speedup
  large-function-insns
  large-function-growth
  large-unit-insns
  inline-unit-growth
  max-inline-insns-recursive
  max-inline-insns-recursive-auto
...
```

Function inlining under code size constraints

- Consider a call graph CG = (V,E,B,C,D), with a node for every function, an edge $e=(f_i,f_k)$ if f_i calls f_k and
  - $B(f_i)$ the code size of f_i w/o inlining
  - $C(e)$ the number of static calls of f_k in f_i
  - $D(f_i)$ the number of dynamic calls to f_i

- **Problem formulation:** Inline a set of functions so that performance is optimized while keeping the code size below a threshold L
  - Not trivial due to **mutual dependence** between inlining of different functions

Function inlining: Formulation

- Given a CG=(V,E,B,C,D), let $b_i = \begin{cases} 0 & f_i \ not \ inlined \\ 1 & f_i \ inlined \end{cases}$
- Find inlining $B = (b_1, \ldots, b_{|V|}) \in \{1,0\}^{|V|}$ such that $D(B) = \sum_{i:b_i=0} D(f_i)$ is minimized, subject to a size constraint

$$S(B) = \sum_{i=1}^{|V|} S(f_i) < T,$$

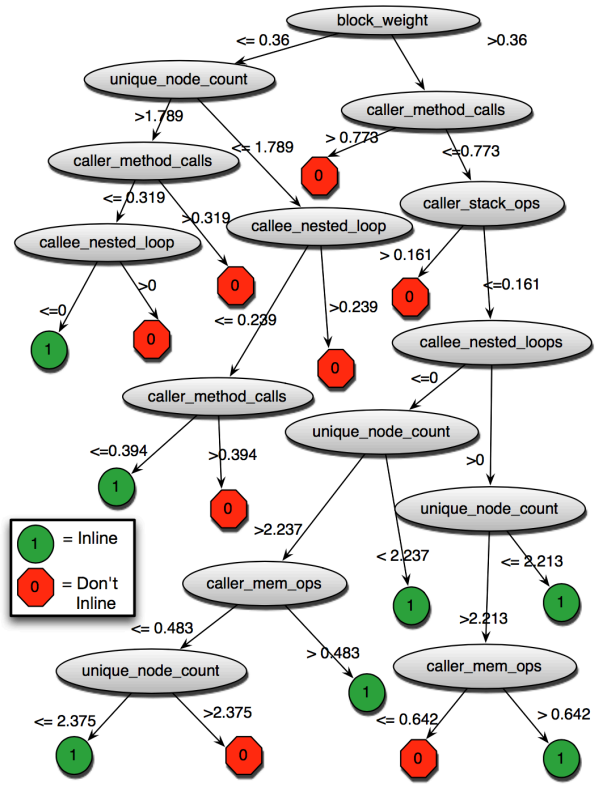
$$S(f_i) = B(f_i) + \sum_{j:b_j=1} C(f_i, f_j) \cdot S_j$$

Assumption: If inlined, then for all static call sites at once

Recursive computation (does not support cyclic call graphs)

Solution approaches

- Branch & bound solution: Expensive but amortizable for embedded applications
  - R. Leupers and P. Marwedel, "Function inlining under code size constraints for embedded processors," ICCAD'99
- Auto-tuning for dynamically compiled languages
  - J. Cavazos and M. F. P. O'Boyle, "Automatic Tuning of Inlining Heuristics," SC'05
- Using machine learning
  - S. Kulkarni, J. Cavazos, C. Wimmer and D. Simon, "Automatic construction of inlining heuristics using machine learning," CGO'13



Source: S. Kulkarni, et al, "Automatic construction of inlining heuristics using machine learning," CGO'13

Parallelization

Theorem (Allen/Kennedy): Any reordering transformation that preserves every dependence in a program preserves the meaning of that program

"Optimizing Compilers for Modern Architectures: A Dependence-Based Approach", Allen and Kennedy, 2002, Ch. 2.

➔ You can rewrite your program as long as you do not violate the dependencies

Restricted case: Static affine nested loops

```
for (i = 0; i < N; ++i)
  for (j = i; j < N; ++j)
    A[2i + 3][4j] = i * j;
```

- Polyhedral compilation
  - Represent loop execution: one point per iteration execution
  - Compact representation, yet possible to reconstruct total order
  - Deal with: parametric and infinite domains, and non-unit stride
  - Reason about: transformation, locality and schedule

Requires precise dependence analysis

- Recall: RAW, WAW, WAR and RAR (for re-use)
- Traditional in compilers: at the granularity of objects (=array)
- Polyhedral: at the granularity of array cells
- Example

```
s0: for (i=2; i<=4; i++) {
s1:   b[i] = a[3i-5] + 2;
s2:   a[2i+1] = 100;}
```

- Is there a dependency between the statements within an iteration?
  - ➔ Solve $3i-5 = 2i+1$ ➔ $i = 6$, but $6 \notin [2,4]$ ➔ No dependency!
- Across iterations: integer solution to $3i_1-5 = 2i_2+1$?
  - ➔ Pairs ($i_1 = 2k$, $i_2 = 3k-3$) are valid solutions
  - ➔ Solution $k = 2$: $(4,3) \in [2,4]$ ➔ **RAW on a[7]** (written in it. 3, read in it. 4)

Dependency analysis in loops

- Analyze accesses of the form $A[i_1,i_2,\ldots,i_n]$, $A[k_1,k_2,\ldots,k_n]$
- General case: Arbitrary index expressions, e.g., function calls: **undecidable**
- Decidable for affine index expressions:
  - $i_h = c_{h1}\, x_1 + \ldots + c_{hm}\, x_m + c_h$
  - $k_h = d_{h1}\, x_1 + \ldots + d_{hm}\, x_m + d_h$

➔ Need to solve a constrained **diophantine** system of equations (inter-iteration)

- $i_1 = k_1$ ➔ $c_{11}\, x_1 + \ldots + c_{1m}\, x_m + c_1 = d_{11}\, y_1 + \ldots + d_{1m}\, y_m + d_1$
- $i_2 = k_2$ ➔ $c_{21}\, x_1 + \ldots + c_{2m}\, x_m + c_2 = d_{21}\, y_1 + \ldots + d_{2m}\, y_m + d_2$
- ...
- $i_n = k_n$ ➔ $c_{n1}\, x_1 + \ldots + c_{nm}\, x_m + c_n = d_{n1}\, y_1 + \ldots + d_{nm}\, y_m + d_n$
- $low_i \leq x_i, y_i \leq high_i$ for $i = 1 \ldots n$

Equivalent to solving integer linear programs (NP complete)

Polyhedral model

- Idea:
  - Index functions: as affine functions $A\left(f_A\left(x_A\right)\right) : f_A(x_A) = F_A \cdot x_A + a_A$
  - Loop bounds: affine inequalities $x_A \in D_A \Rightarrow G_A \cdot x_A + b_A \geq 0$

- A statement $A(x_A)$ depends on statement $B(x_B)$, denoted $(B(x_B) \rightarrow A(x_A))$, if
  - $A(x_A)$ and $B(x_B)$ refers to the same memory location M
  - Indexes in the iteration spaces: $x_A \in D_A$, $x_B \in D_B$
  - In original sequential order $B(x_B)$ is executed before $A(x_A)$

Polyhedral compilation: Dependence polyhedron

- A statement $A(x_A)$ depends on statement $B(x_B)$, denoted $(B(x_B) \rightarrow A(x_A))$, if
  - $A(x_A)$ and $B(x_B)$ refers to the same memory location M

$$f_A(x_A) = f_B(x_B) \rightarrow F_A \cdot x_A + a_A = F_B \cdot x_B + a_B$$

  - Indexes in the iteration spaces: $x_A \in D_A$, $x_B \in D_B$

$$G_A \cdot x_A + b_A \geq 0 \wedge G_B \cdot x_B + b_B \geq 0$$

  - In original sequential order $B(x_B)$ is executed before $A(x_A)$
    - For nesting level l:

$$i < l : x_{A,i} = x_{B,i},\ x_{A,i} > x_{B,i} \text{ otherwise}$$
$$\rightarrow P_A \cdot x_A - P_B \cdot x_B + c \geq 0$$

Polyhedral compilation: Dependence polyhedron (2)

- Putting all together: The dependence polyhedron between the instances and a given nesting level is:

$$\mathcal{D}_{B,A,l} : \left[\begin{array}{cc} F_A & -F_B \\ \hline G_A & 0 \\ 0 & G_B \\ P_A & P_B \end{array}\right] \cdot \begin{bmatrix} x_A \\ x_B \end{bmatrix} + \left[\begin{array}{c} a_A - a_B \\ \hline b_A \\ b_B \\ c \end{array}\right] \begin{array}{c} = 0 \\ \hline \geq 0 \end{array}$$

  - There is a dependency edges from every two iterations x_A and x_B for which the equation above holds!

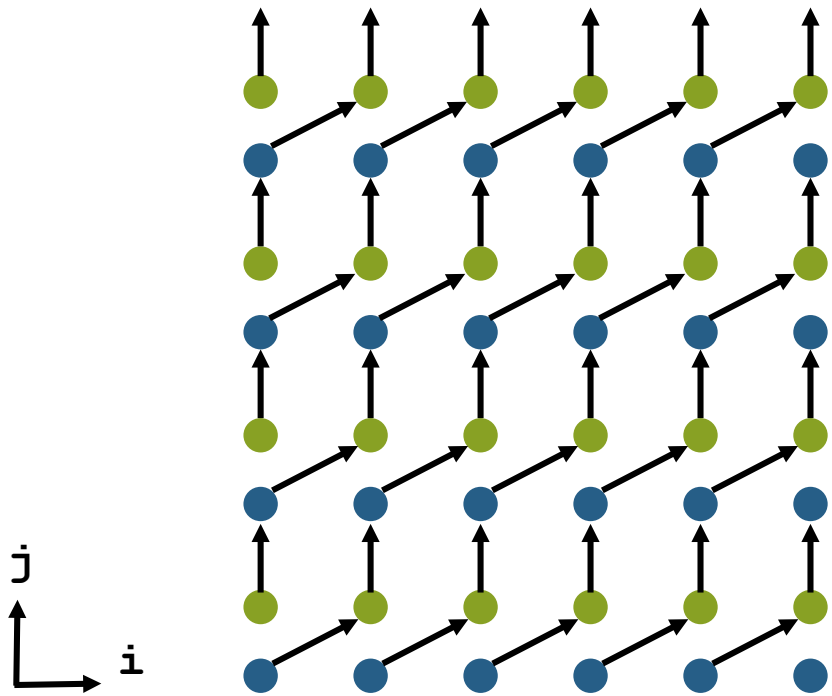
Partitioning: Intuition

- Example

```
        for (i = 1; i <= 100; i++)
            for (j = 1; j <= 100; j++) {
S1:           X[i][j] = X[i][j] + Y[i-1][j];
S2:           Y[i][j] = Y[i][j] + X[i][j-1];
            }
```

Parallelism?

- Dependence polyhedron

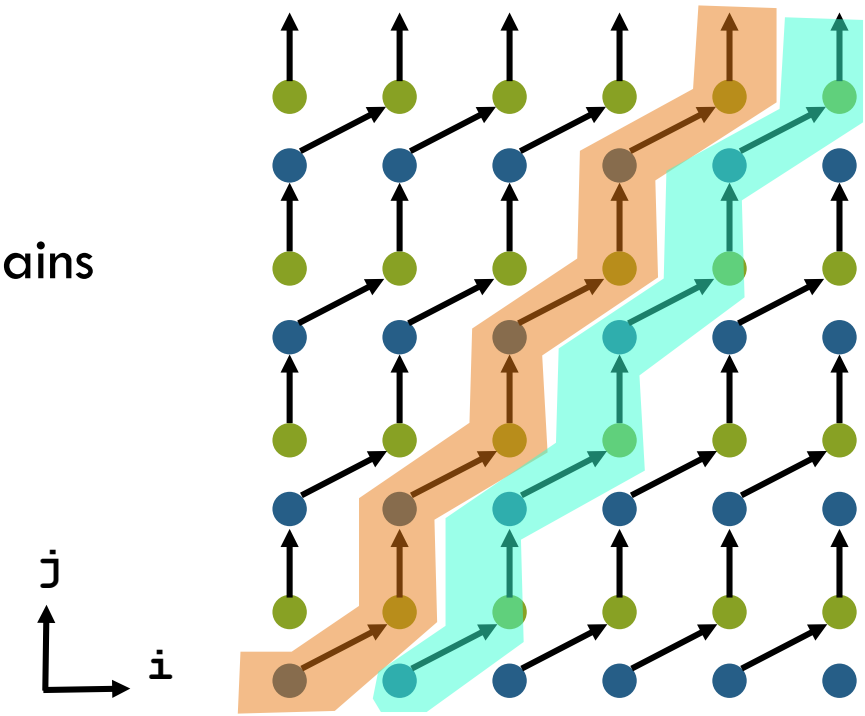


Partitioning: Intuition (2)

- Example

```
        for (i = 1; i <= 100; i++)
            for (j = 1; j <= 100; j++) {
S1:           X[i][j] = X[i][j] + Y[i-1][j];
S2:           Y[i][j] = Y[i][j] + X[i][j-1];
            }
```

- Data-level parallelism?
- Execute independent chains



How many chains?

Is this a good idea?

Partitioning: Representation

- Simple case: **Affine partition** for **synchronization-free** parallelism
- Affine partition for an access A is an affine mapping to the **processor space**

$$C_A \cdot x_A + c_A$$

  - It maps every statement to a processor index (could be 1D, 2D, …)
- Given two dependent accesses $B(x_B) \rightarrow A(x_A)$, at least one is a write

$$f_A(x_A) = f_B(x_B) \rightarrow F_A \cdot x_A + a_A = F_B \cdot x_B + a_B$$

$$G_A \cdot x_A + b_A \geq 0 \wedge G_B \cdot x_B + b_B \geq 0$$

- The partition is synchronization free if $C_A \cdot x_A + c_A = C_B \cdot x_B + c_B$
- Potential many partitions: Pick the one with the highest rank
  - Intuition: The highest the rank the more processors are used (potentially more parallelism). A matrix with a high rank "distributes" better

Partitioning: Example

```
        for (i = 1; i <= 100; i++)
            for (j = 1; j <= 100; j++) {
S1:           X[i][j] = X[i][j] + Y[i-1][j];
S2:           Y[i][j] = Y[i][j] + X[i][j-1];
            }
```

Need two partitions (one per statement)

- Partition for 1D array of processors $\begin{bmatrix} C_{11} & C_{12} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + c_1 = \begin{bmatrix} C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + c_2$

- Dependency on X between S1 and S2? `i = i'` `j = j'-1`
- Dependency on Y between S1 and S2? `i = i'+1` `j = j'`
- Replacing in the partition equation

$$\begin{bmatrix} C_{11} - C_{21} & C_{12} - C_{22} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + c_1 - c_2 - C_{22} = 0 \qquad \begin{bmatrix} C_{11} - C_{21} & C_{12} - C_{22} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + c_1 - c_2 + C_{21} = 0$$

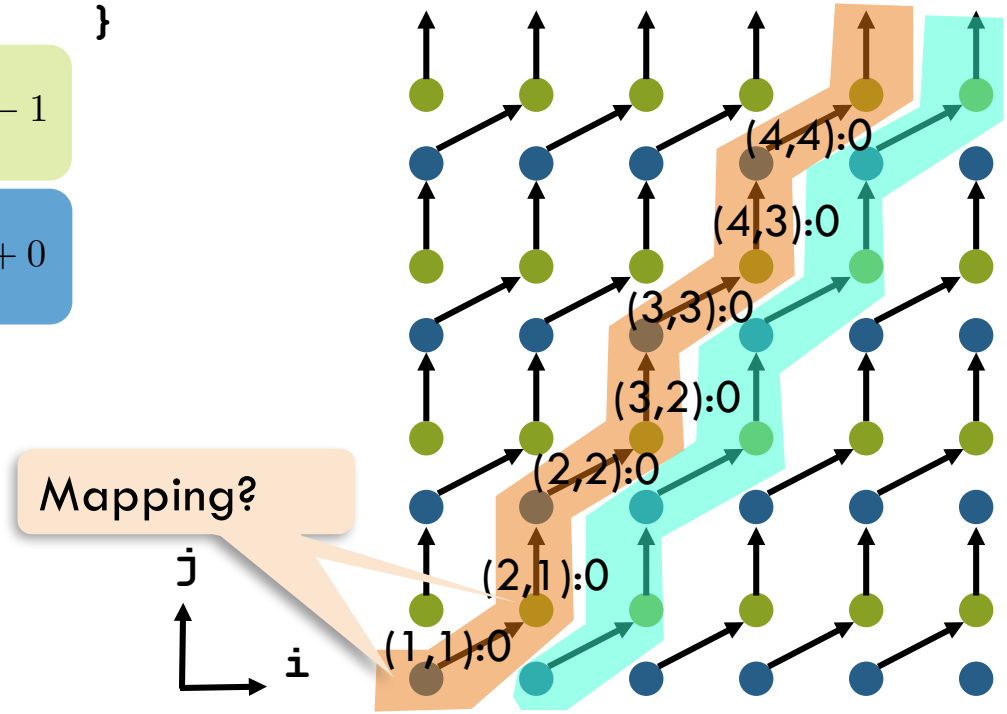
- Solution $S1 : p = \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} - 1 \qquad S2 : p = \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + 0$

Partitioning: Example (2)

```
        for (i = 1; i <= 100; i++)
            for (j = 1; j <= 100; j++) {
S1:           X[i][j] = X[i][j] + Y[i-1][j];
S2:           Y[i][j] = Y[i][j] + X[i][j-1];
            }
```

$$S1 : p = \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} - 1$$

$$S2 : p = \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + 0$$



Code generation

- Simple code generation: Iterate over the processor space

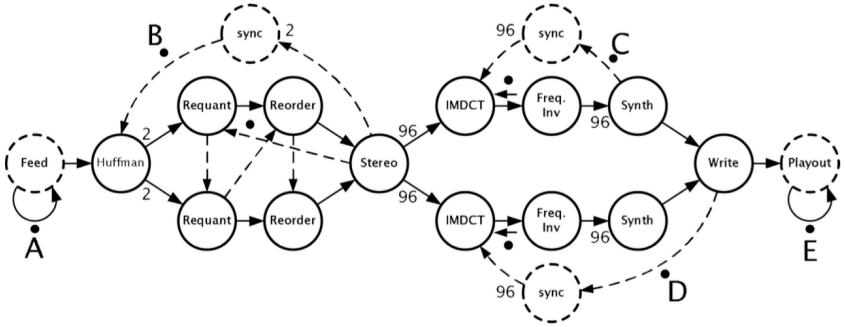
```
for (p = -100; p <= 99; p++)
   for (i = 1; i <= 100; i++)
      for (j = 1; j <= 100; j++) {
        if (p == i-j-1) X[i][j] = X[i][j] + Y[i-1][j];
        if (p == i-j)   Y[i][j] = Y[i][j] + X[i][j-1];
      }
```

- Efficiency?
  - Each processor executes at most 99 iterations, but iterates over 100x100!
  - Each iteration includes now control statements

- Many more optimization possibilities

Dataflow programming

- Start already with a parallel model
- Dataflow: Application as a graph of parallel actors that communicate solely via channels



MP3 decoder

Source: Geilen, Marc. "Synchronous dataflow scenarios." *ACM Transactions on Embedded Computing Systems (TECS)* 10.2 (2010): 16.

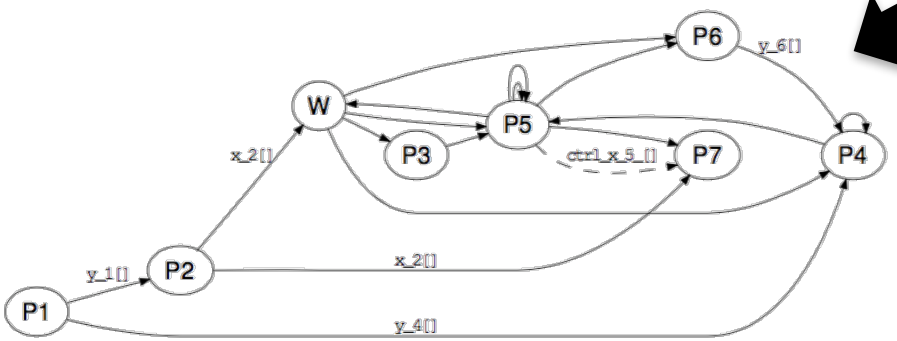
- Multiple variants of dataflow models
  - Different rules to determine if an actor executes (aka **fires**)
  - Different number of allowed behaviors

- **Static:** Firing behavior known at compile-time
- **Dynamic:** Firing behavior may depend on data

Dataflow and polyhedral compilation

- It is sometimes possible to extract a dataflow representation from sequential code
- Using polyhedral analysis
  - N. Dmitry "Automatic derivation of polyhedral process networks from while-loop affine programs" ESTIMedia'11
- Using dynamic analysis on C programs
  - M. Aguilar "Extraction of Kahn Process Networks from While Loops in Embedded Software." *HPCC'15*

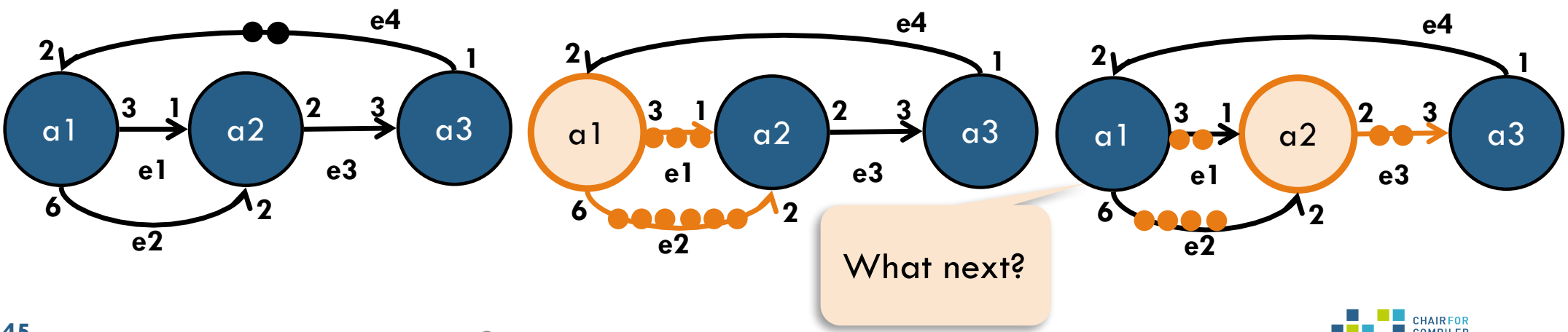
```
#parameter EPS 0.005
w = 0
ctrl_x_5 = (N+1,0)
for i = 1 to N,
S1:  y_1[i] = F1()
     in_2 = y_1[i]
S2:  x_2[i] = F2( in_2 )
W    while (in_w = σx(⟨W,(i,w)⟩) >= EPS),
       w = w + 1
S3:    x_3[i,w] = F3()
       for j = i+1 to N+1,
         in_4 = σy(⟨S4,(i,w,j)⟩)
S4:      y_4[i,w,j] = F4( in_4 )
         in_5_x = σx(⟨S5,(i,w,j)⟩)
         in_5_y = y_4[i,w,j]
S5:      x_5[i,w,j] = F5( in_5_x, in_5_y )
         ctrl_x_5 = (i,w)
       endfor
       in_6 = σx(⟨S6,(i,w)⟩)
S6:    y_6[i,w] = F6( in_6 )
     endwhile
     ctrl_x_5_[i] = ctrl_x_5

     (α,β) = ctrl_x_5_[i]
     in_7 = σx(⟨S7,(i,α,β)⟩)
S7:  out = F7( in_7 )
endfor
```



Synchronous dataflow graph (SDF)

Def.: An **SDF** is an annotated multi-graph G=(V,E,W). V is the set of actors and E $\subseteq$ VxV the set of channels. W = $\{w_1,\ldots,w_{|E|}\} \subset N^3$ is a set of annotations for every channel e = (a_1,a_2), $w_e=(p_e,c_e,d_e)$: p_e is the number of tokens produced by a firing of a_1, c_e the tokens consumed by a_2 and d_e the amount of delay tokens

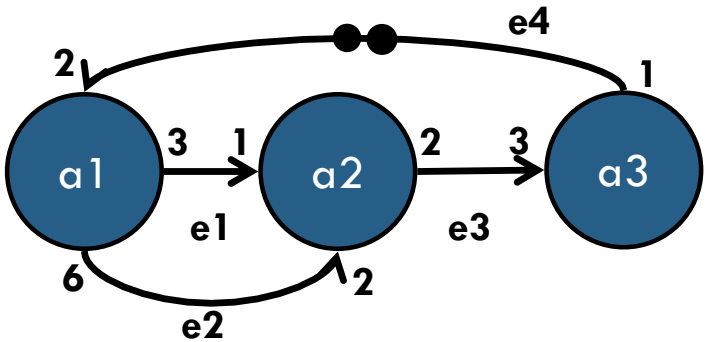
- Example



Topology matrix

Def.: Given an SDF G=(V,E,W), its **topology matrix Γ** has a row for every channel and a column for every actor. $[\Gamma_{ik}] = p_{ik} - c_{ik}$ (i.e. the amount of tokens produced minus those consumed by actor k on/from channel i)

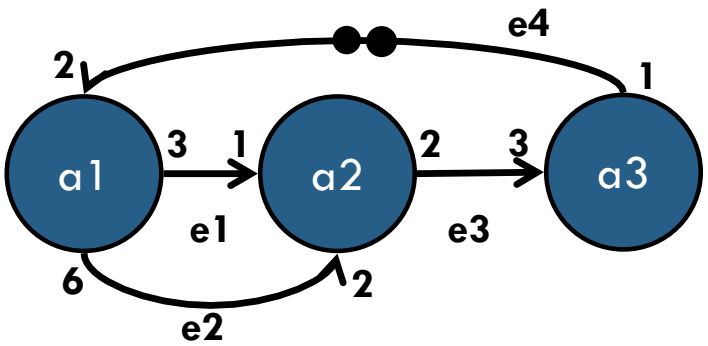
- Example

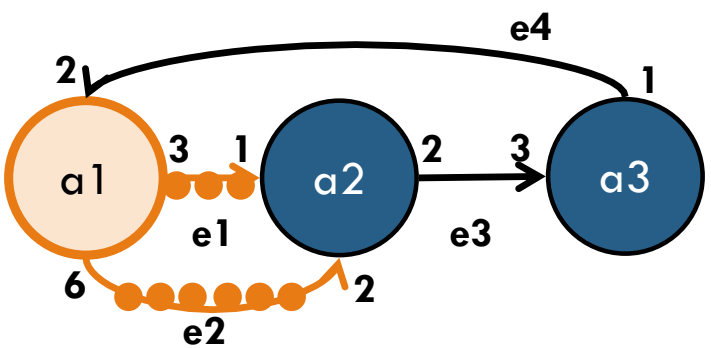


$$\Gamma = \begin{pmatrix} 3 & -1 & 0 \\ 6 & -2 & 0 \\ 0 & 2 & -3 \\ -2 & 0 & 1 \end{pmatrix}$$

Topology matrix (2)

- **State** of an SDF: tokens in the queues
- The topology matrix can be used to update the state after a firing



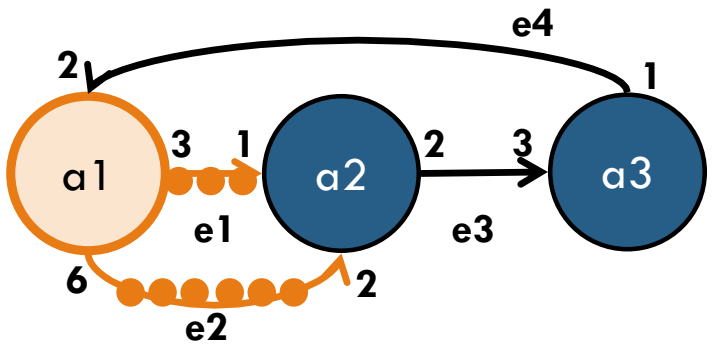


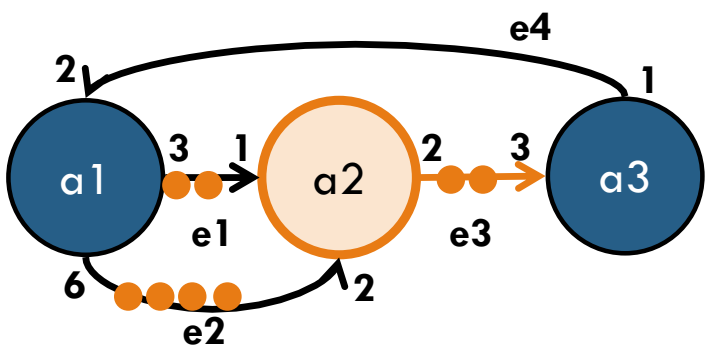
$$\Gamma = \begin{pmatrix} 3 & -1 & 0 \\ 6 & -2 & 0 \\ 0 & 2 & -3 \\ -2 & 0 & 1 \end{pmatrix}$$

$$s_1 = s_0 + \begin{pmatrix} 3 & -1 & 0 \\ 6 & -2 & 0 \\ 0 & 2 & -3 \\ -2 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 2 \end{pmatrix} + \begin{pmatrix} 3 \\ 6 \\ 0 \\ -2 \end{pmatrix} = \begin{pmatrix} 3 \\ 6 \\ 0 \\ 0 \end{pmatrix}$$

Topology matrix (3)

- **State** of an SDF: tokens in the queues
- The topology matrix can be used to update the state after a firing





$$\Gamma = \begin{pmatrix} 3 & -1 & 0 \\ 6 & -2 & 0 \\ 0 & 2 & -3 \\ -2 & 0 & 1 \end{pmatrix}$$

$$s_2 = s_1 + \begin{pmatrix} 3 & -1 & 0 \\ 6 & -2 & 0 \\ 0 & 2 & -3 \\ -2 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ 6 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -1 \\ -2 \\ 2 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 2 \\ 0 \end{pmatrix}$$

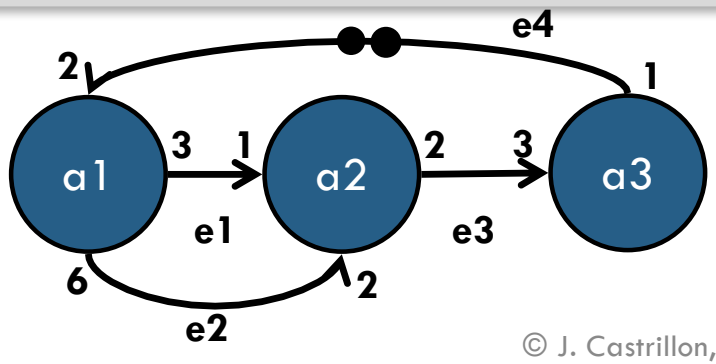
Complete cycle and repetition vector

Def.: Given an SDF G=(V,E,W), a **complete cycle** is a sequence of actor firings that brings the SDF to its initial state

$$s_n = s_0 + \Gamma \cdot (v_1 + v_2 + \cdots + v_n) = s_0$$

$$\Gamma \cdot (v_1 + v_2 + \cdots + v_n) = \Gamma \cdot \vec{r} = 0$$

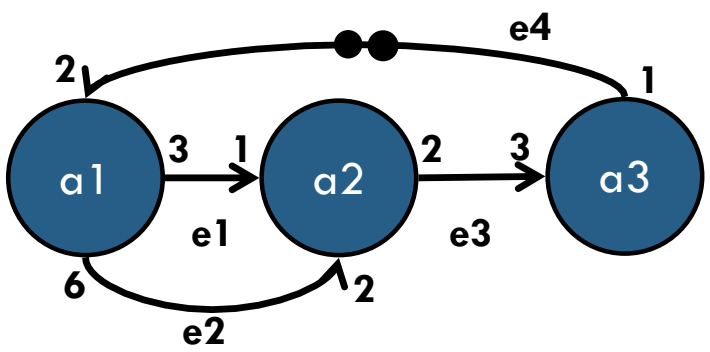
Def.: Given an SDF with topology matrix $\mathbf{\Gamma}$, the **repetition vector** is the smallest integer vector in the null space of $\mathbf{\Gamma}$, i.e., $\Gamma \cdot \vec{r} = 0$



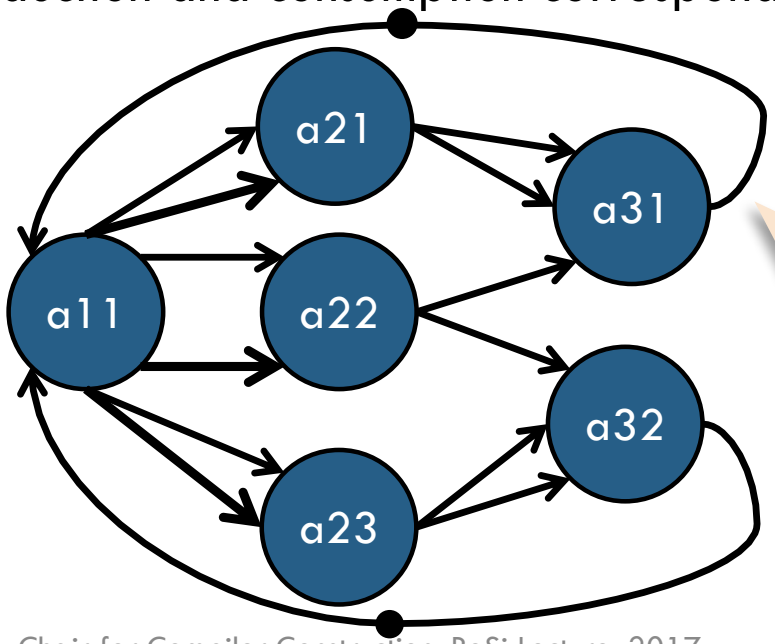
$$\begin{pmatrix} 3 & -1 & 0 \\ 6 & -2 & 0 \\ 0 & 2 & -3 \\ -2 & 0 & 1 \end{pmatrix} \cdot \vec{r} = 0 \quad \Rightarrow \quad \vec{r} = (1, 3, 2)^T$$

Unrolling an SDF

- Intuition
  - Given a repetition vector $r=(q_1,q_2,\ldots q_n)$
  - Make q_i copies of actor a_i (need to adapt the ports)
  - Connect actors so that production and consumption correspond to original SDFG



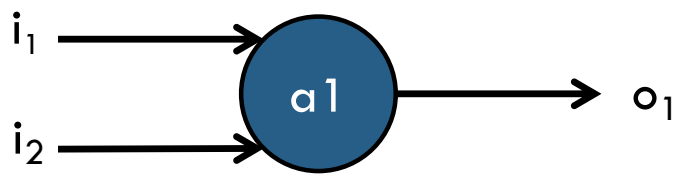
$$\vec{r} = (1, 3, 2)^T$$



Can be done more times to expose more parallelism

Analysis

- For static models: Possible to analyze deadlock freedom and max throughput



- Notation
  - I_a, O_a: Set of inputs an ouputs of actor a
  - $t_k(i)/t_k(o)$: Time of arrival/production of k-th token in input i, or output o
  - $e_k(a)$: k-th enabling of actor a
- Enabling

$$e_k(a) = \max_{i \in I_a}(t_k(i))$$

- **Starting time** $s_k(a)$ can be anytime **after enabling** and **finish time,** $f_k(a)$, after a duration $\rho_k(a)$

Recall: Self-timed execution of HSDFs

- Zero-delay
  - Token consumption: atomically at $s_k(a)$
  - Token production: atomically at $f_k(a)$
- Self-time execution: Actor starts as soon as data is there

$$\forall o \in O_a,\ t_k(o) = \rho_k(a) + \max_{i \in I_a}(t_k(i))$$

- Finish times can be expressed as a system of equations in **max-plus algebra**

Mapping to heterogeneous multicores

- For static models
  - Extract directed acyclic graph (DAG) from unrolled graph and apply DAG-mapping algorithms (very common in compilers and scheduling theory), or
  - Work on cyclic graphs, e.g., with modulo scheduling (used for SW-pipelining)
  - Overlapped schedules are fundamentally superior than blocked schedules with unfolding and retiming

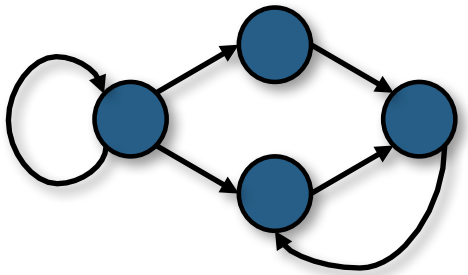
K. K. Parhi and D. G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. IEEE Transactions on Computers, 1991.

- For dynamic models
  - Characterization of events (traces, real-time calculus)

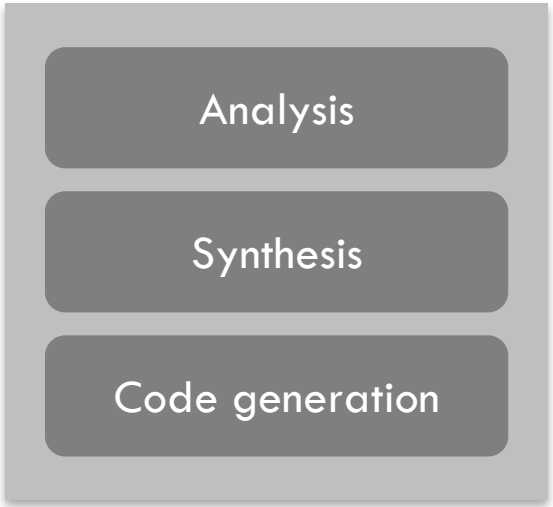
- In any case: Need abstract models of multi-processor architecture

Generic compilation flow

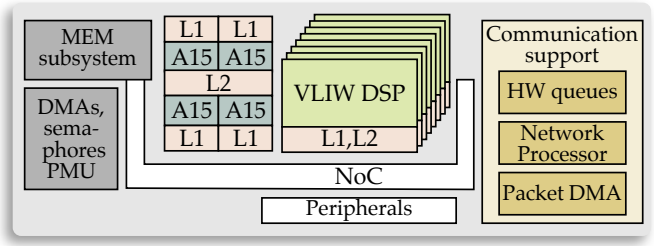
Dataflow Application



Non-functional specification



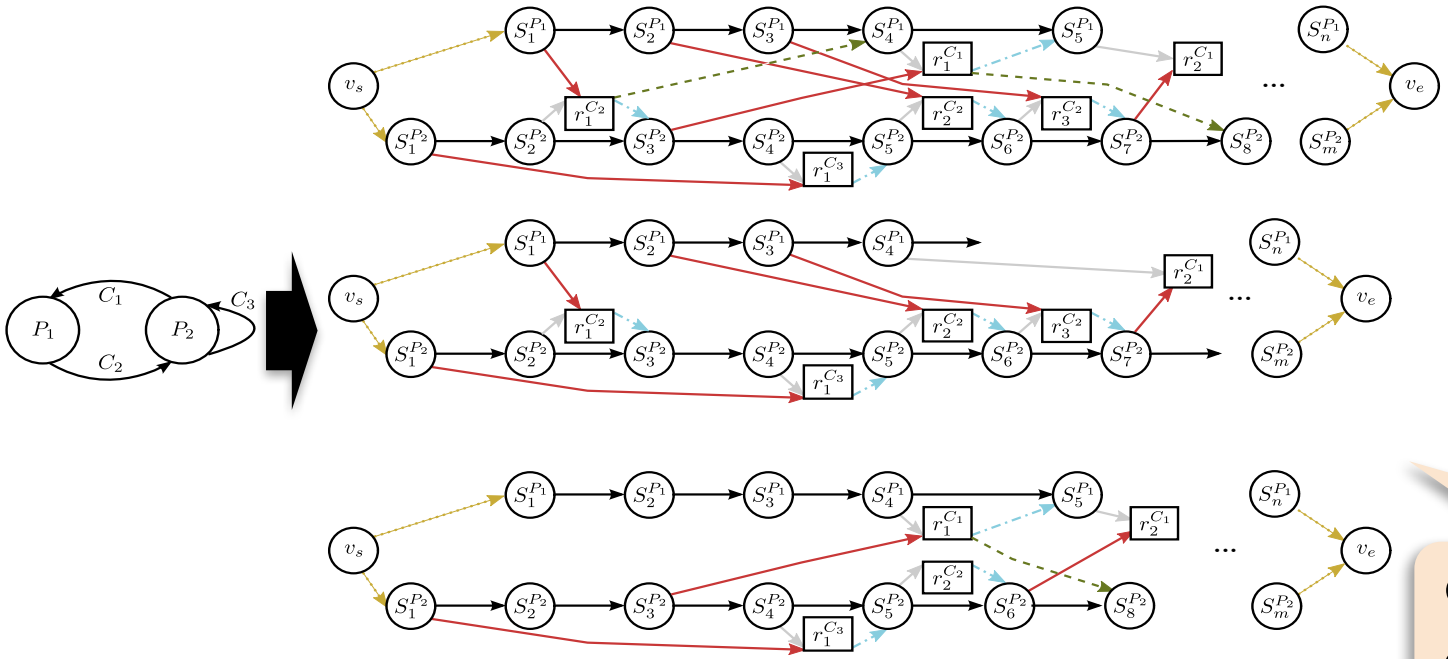
Architecture model

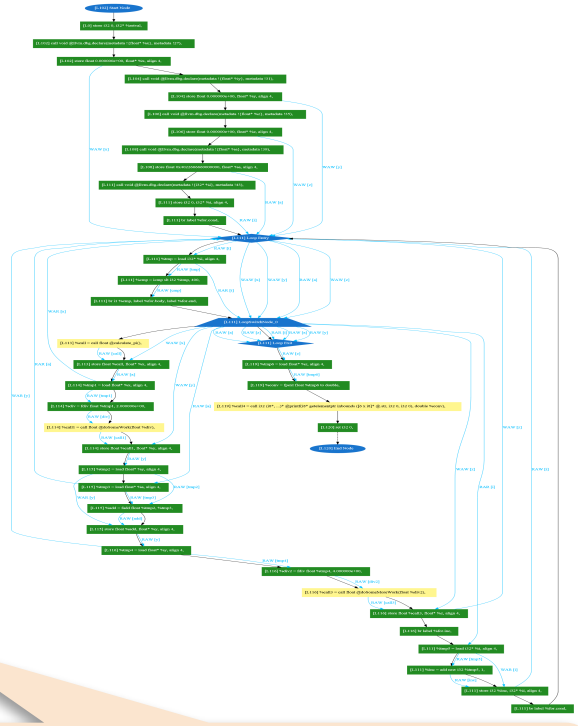



```
 PNargs_ifft_r.ID = 6U;
    PNargs_ifft_r.PNchannel_freq_coef = filtered_coe
    PNargs_ifft_r.PNnum_freq_coef = 0U;
    PNargs_ifft_r.PNchannel_time_coef = sink_right;
    PNargs_ifft_r.channel = 1;
    sink_left = IPCllmrf_open(3, 1, 1);
    sink_right = IPCllmrf_open(7, 1, 1);
    PNargs_sink.ID = 7U;
    PNargs_sink.PNchannel_in_left = sink_left;
    PNargs_sink.PNnum_in_left = 0U;
    PNargs_sink.PNchannel_in_right = sink_right;
    PNargs_sink.PNnum_in_right = 0U;
    taskParams.arg0 = (xdc_UArg)&PNargs_src;
    taskParams.priority = 1;

ti_sysbios_knl_Task_create((ti_sysbios_knl_Task_Func
&taskParams, &eb);
    glob_proc_cnt++;
    hasProcess = 1;
    taskParams.arg0 = (xdc_UArg)&PNargs_fft_l;
    taskParams.priority = 1;

ti_sysbios_knl_Task_create((ti_sysbios_knl_Task_Func
ft_Templ, &taskParams, &eb);
    glob_proc_cnt++;
    hasProcess = 1;
    taskParams.arg0 = (xdc_UArg)&PNargs_ifft_r;
    taskParams.priority = 1;

ti_sysbios_knl_Task_create((ti_sysbios_knl_Task_Func
fft_Templ, &taskParams, &eb);
    glob_proc_cnt++;
    hasProcess = 1;
    taskParams.arg0 = (xdc_UArg)&PNargs_sink;
    taskParams.priority = 1;
```

Internal model for analysis

- Intermediate representation of actor implementation
- Trace of events, and dependencies among them





Optimize placement of code and data to platform

A. Goens, et al., "Analysis of Process Traces for Mapping Dynamic KPN Applications to MPSoCs", In IFIP International Embedded Systems Symposium (IESS), 2015, Foz do Iguaçu, Brazil (Accepted for publication), 2015

Some sample results

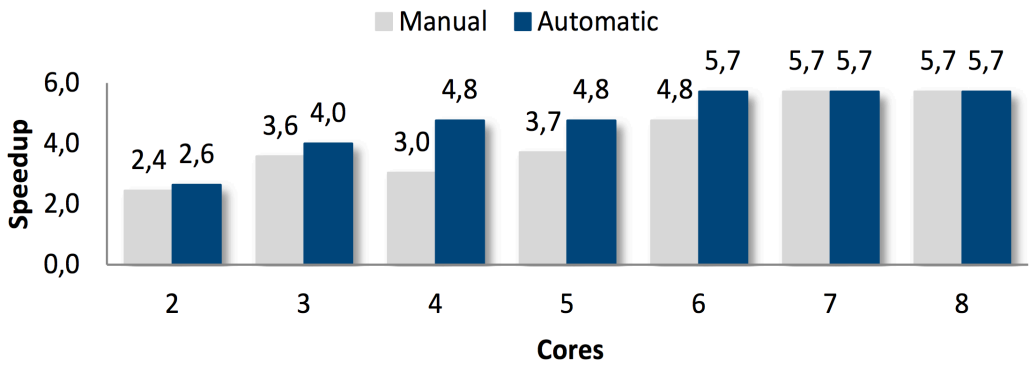
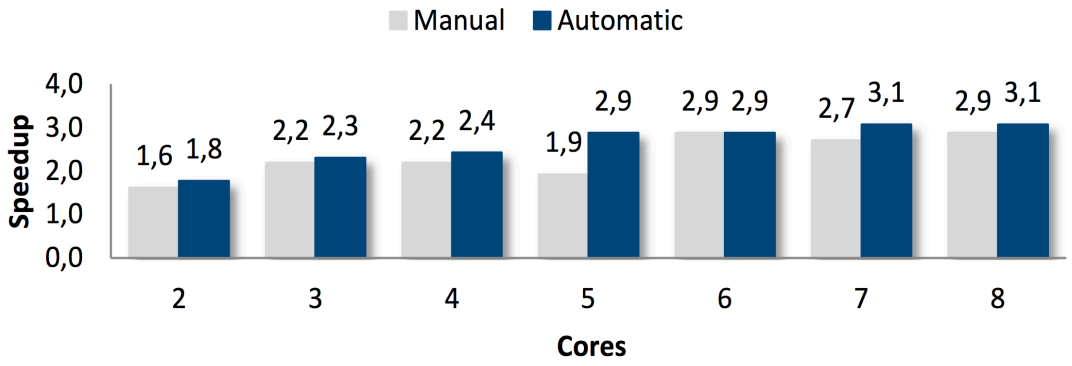
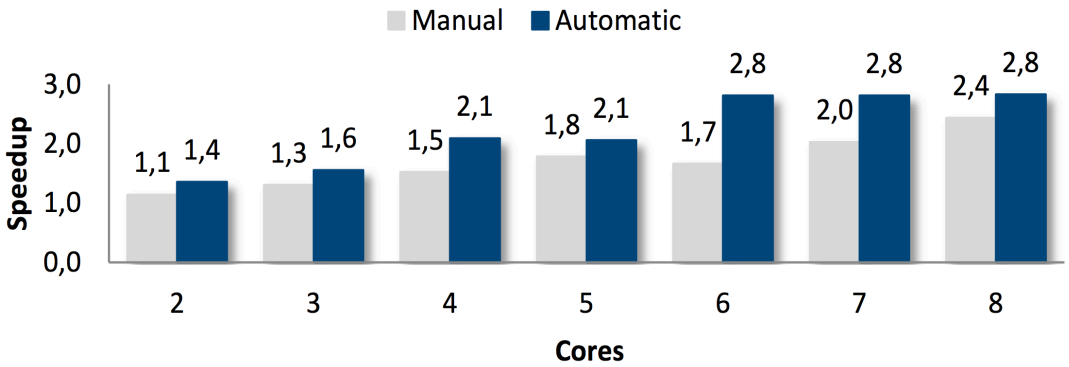


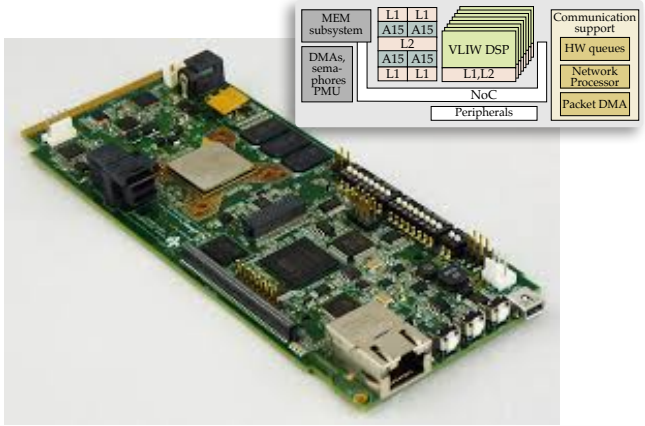
Image processing



Audio filtering application



LTE digital receiver



M. Aguilar, et al., "Improving performance and productivity for software development on TI Multicore DSP platforms," in Education and Research Conference (EDERC), 2014 6th European Embedded Design in , vol., no., pp.31-35, 11-12 Sept. 2014

Agenda

- Introduction
- Sample optimizations
- **Opportunities in role-based programming**

Opportunities in role-based programming

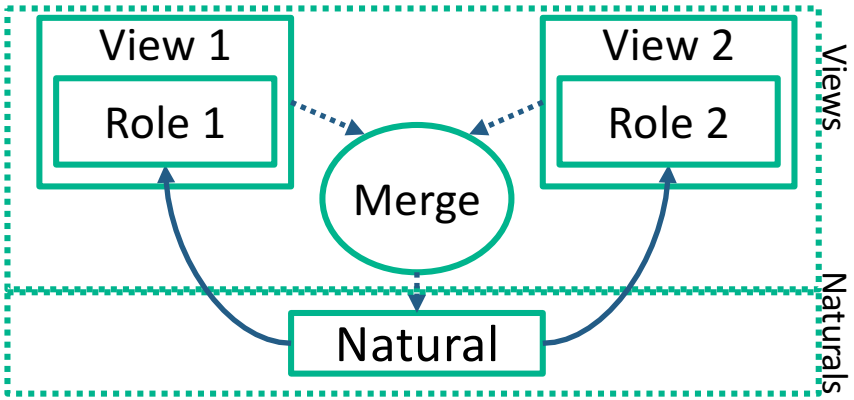
- What changes with RBP?
- Overview of RBP Approaches

What changes with RBP?

- Method dispatch at runtime
  - More indirection
    - 4th dimension in method dispatch (method name, sender, receiver, and context)
    - Multiple methods may be called because of active roles
- Keep part of the model at runtime
  - Played roles, active and inactive contexts
  - Expensive bookkeeping
- Enhanced type system
- Dynamic compilation more important than ever

Overview of RBP approaches

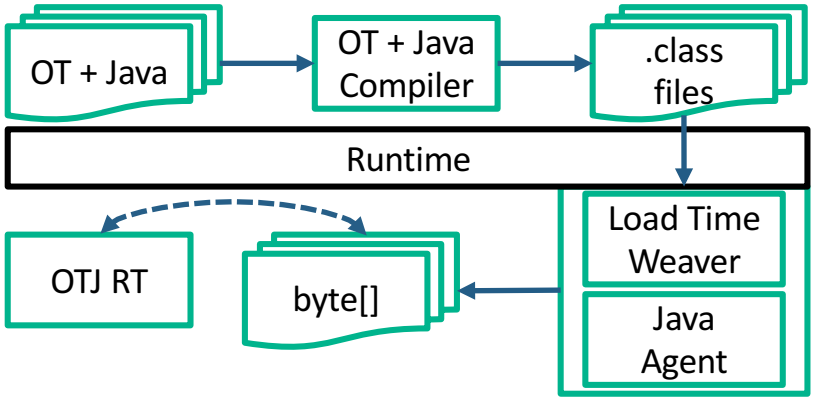
- Object Teams/Java
  - Enhanced Java syntax
  - Roles and Teams as language constructs
  - Implementation via dynamic aspect weaving
- Scroll
  - Scala library
  - View-based Programming
    - Single Underlying Model (SUM)
    - Compartments form views on SUM
- LyRT
  - Java library
  - Subclass-proxys implement management code



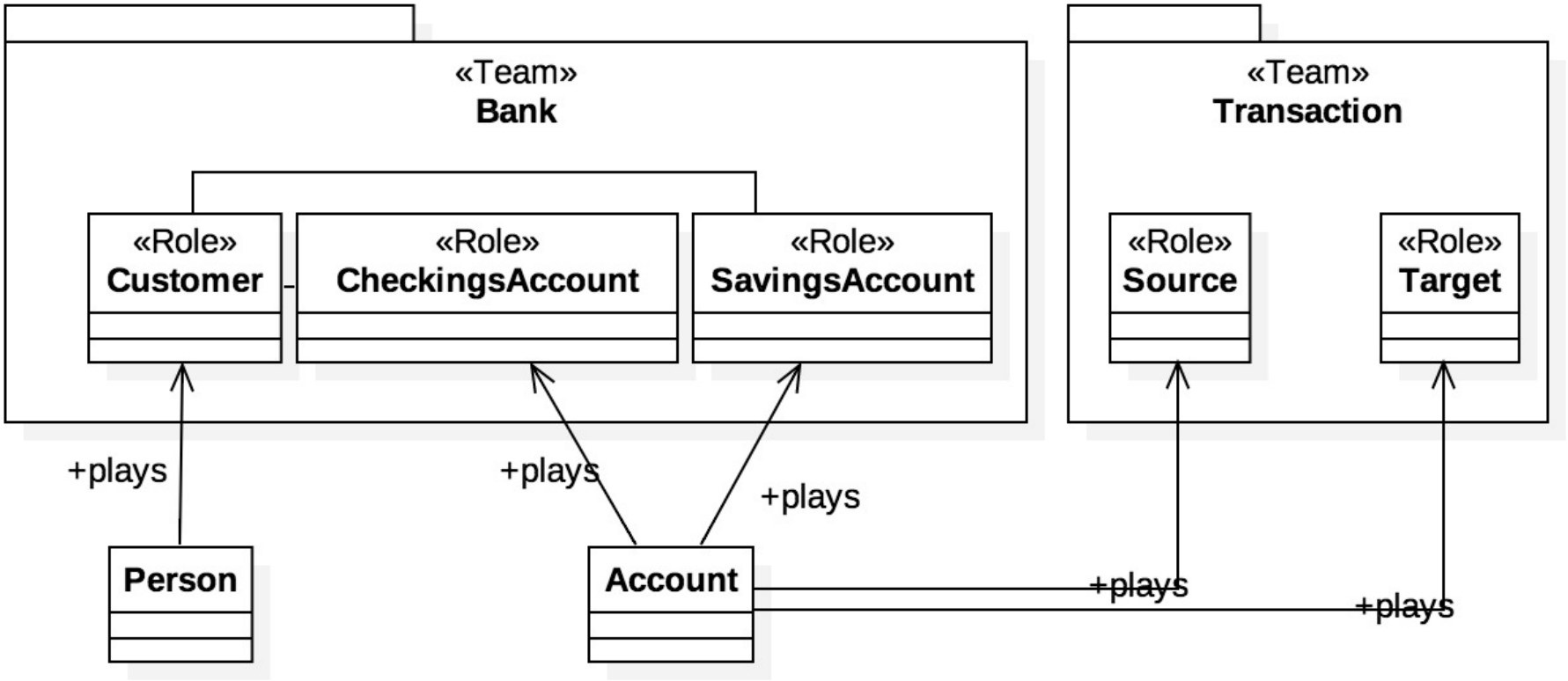
M. Leuthäuser and U. Aßmann. Enabling View-based Programming with SCROLL: Using roles and dynamic dispatch for etablishing view-based programming. MORSE/VAO'15

Object Teams/Java in detail

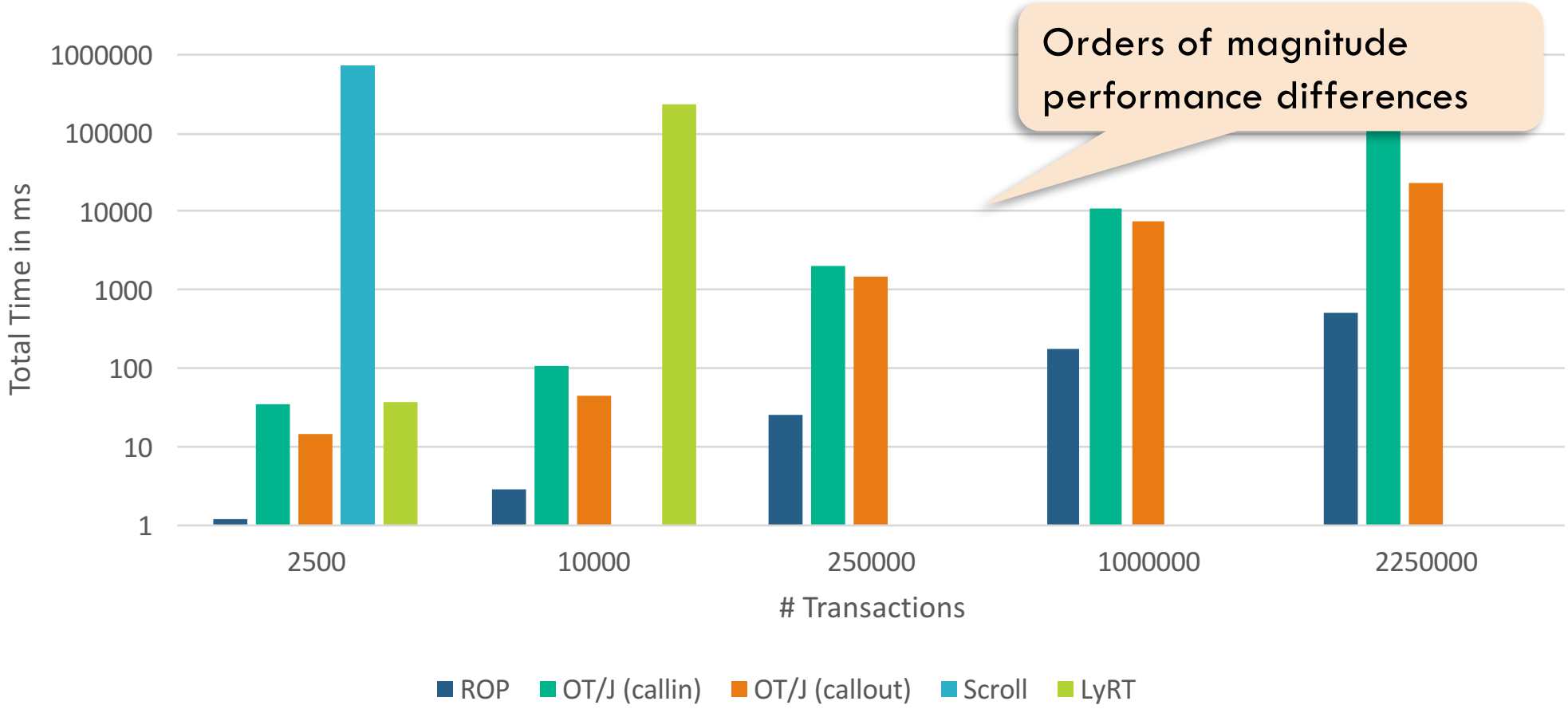
- Extends JDT Compiler
- Aspects to implement role features
  - Before/after/replace callin binding
  - Callout binding
- Weaver rewrite base classes at load time
  - If there is a binding to a base method
- Sources of inefficiencies
  - Bookkeeping of Pointcuts is expensive
    - Garbage collection uses lots of time
  - Chaining to execute all callin bindings expensive



Benchmark

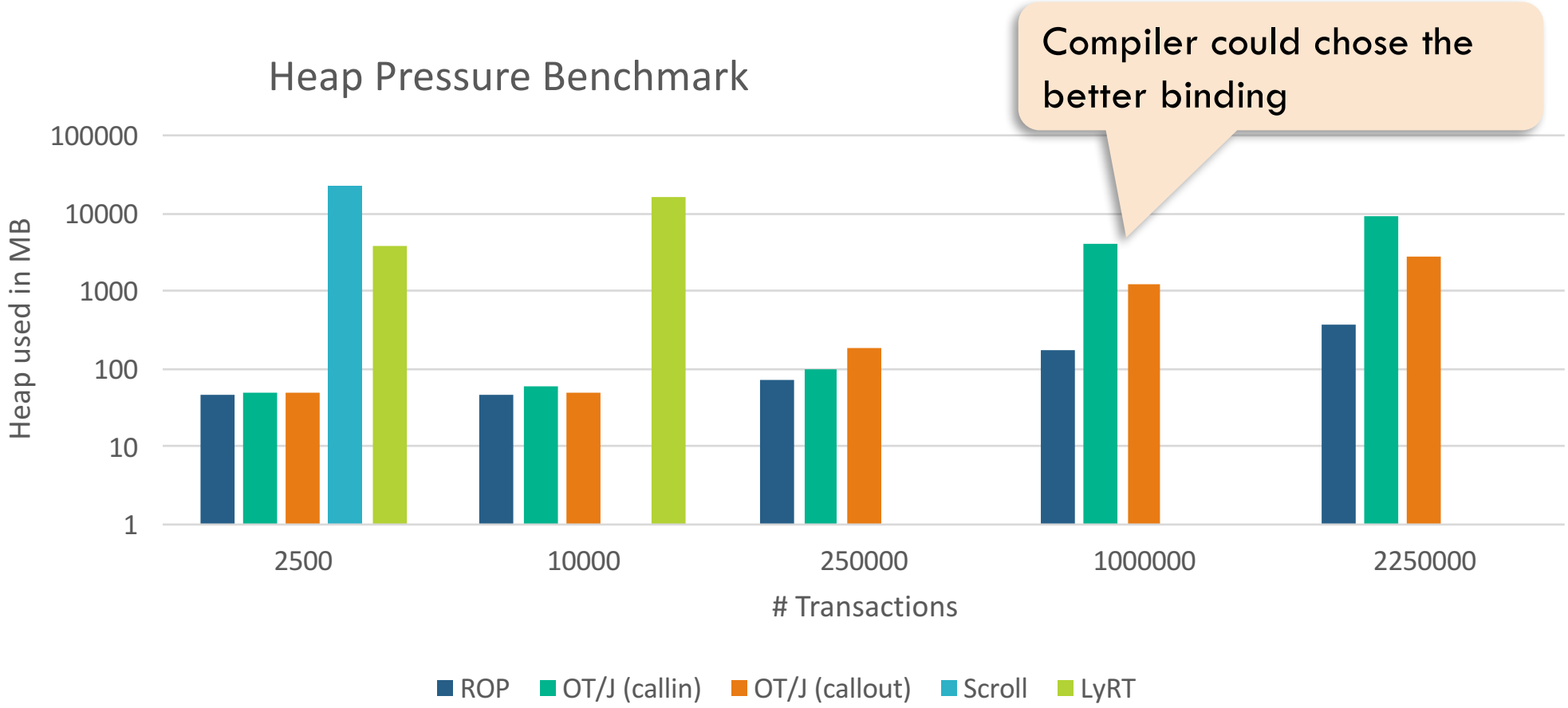


Benchmark: Time



Mac OS 10.11.6, 16GB RAM, JDK 1.8.111, JVM_ARGS="-d64 -Xms1024m -Xmx4048m"

Benchmark: Heap



Mac OS 10.11.6, 16GB RAM, JDK 1.8.111, JVM_ARGS="-d64 -Xms1024m -Xmx4048m"

Closing remarks

- Basic compiler terminology
- Sample compiler optimizations (basic and advanced)
  - Information required
  - Problem characterization
  - Solution approach
- Outlook about performance/memory bottlenecks in current role-based programming

Thanks for you attention
and have fun doing research!!