Compiler Flow for Processors/Systems

Prof. Dr.-Ing. Jeronimo Castrillon

Chair for Compiler Construction

TU Dresden – Cfaed

MPSoC Winter School

Tunis, Tunisia

Nov. 2014

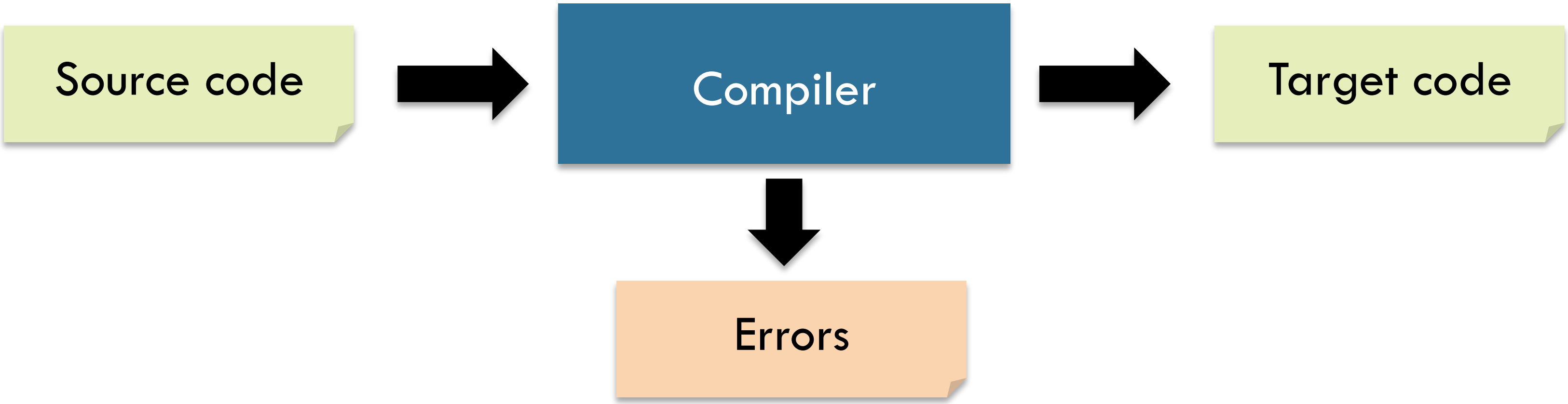
Acknowledgements

- Cfaed content: from the Cfaed colleagues – visit **www.cfaed.tu-dresden.de**
- Basic content: references given when appropriate
- Research content: most of it carried out in Aachen at the Institute for Communication Technologies and Embedded Systems (ICE): Prof. Leupers, Weihua Sheng, Maximilian Odendahl, Miguel Aguilar

1. Introduction

- What is a compiler?
- Why do we care?

What is a compiler



- 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)

History of compilers

- < 1950 Programming in assembly
- 1950s First machine-independent languages (skepticism) – A-0 Language
  To compile: "put things together" – Not really what it is today
- 1959 Complete compiler – J. Backus @ IBM for Fortran (in assembly)
  Complexity: 15 man-year projects

History of compilers (2)

- 1960s Theoretical work for code analysis
- 1970 Bootstrapping becomes mainstream (C-compiler written in C)
  First ever: LISP in 1962
- 1970 Tools to create parts of the compiler (lex & yacc)
- 1980s/90s Code generator generators
- > 2000 Optimizations (mostly backend)

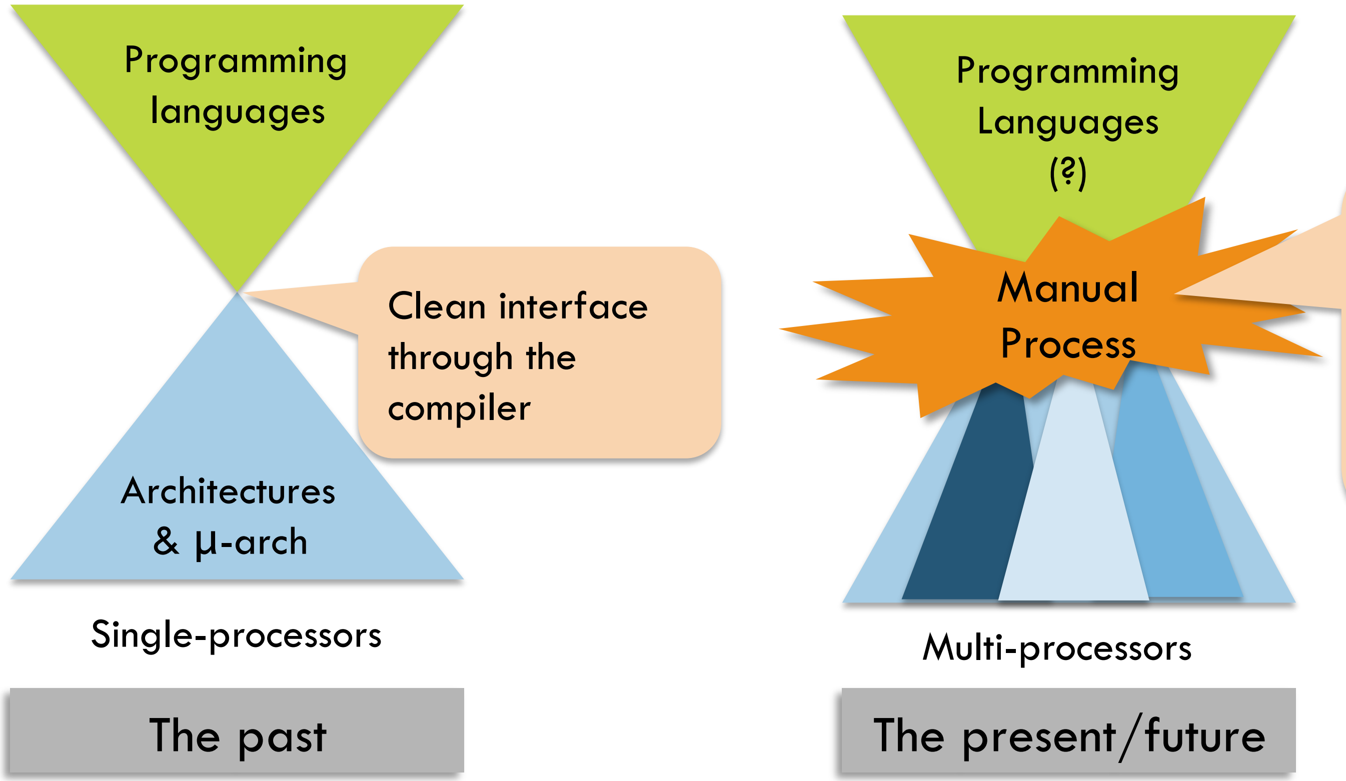
- Today SIMD, vectorization, continuous compilation, new optimization goals, parallelizing compilers, domain-specific languages, skeletons, auto-tuning, …

Compiler goals

- First and foremost: **Correctness**
  - Correct translation while preserving semantics
  - Incorrect code must be discarded

- Optimization goals
  - Typ.: Performance and code size
  - New: Energy/power consumption, robustness
  - Optimality? – Undecidable, often NP complete (especially in the Backend)
- Off-line processing
  - Tolerable turn-around times – time complexity in O(n) – O(n^2)
  - Some domains are more patient (embedded)

Compilers for (heterogeneous) multi/many-cores



- Deal with mapping, scheduling, synchronization
- Different OSes
- Different APIs
- Compile vs. runtime

This lecture

Introduction to classical compilers

Insight into MPSoC compilation
(followed by hands-on)

& µ-arch

Single-processors

The past

Multi-processors

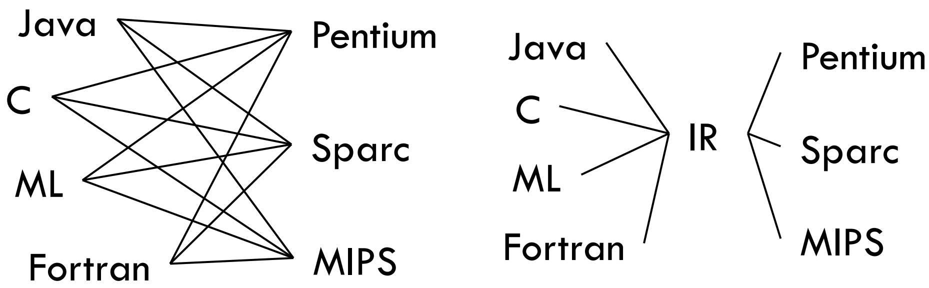
The present/future

2. Classical compilers

- How does a compiler work?
- Learn/refresh compiler phases
- Learn/refresh theoretical background

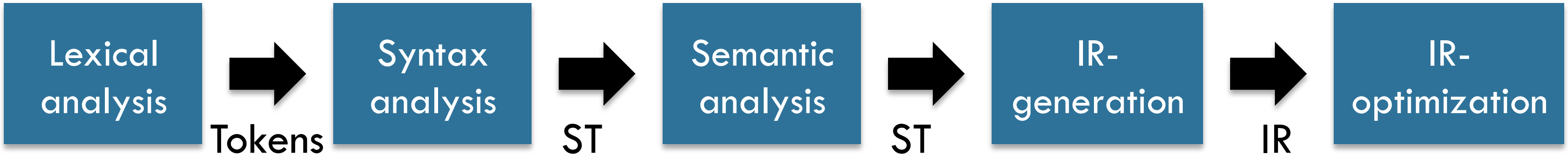
3-phase compiler

- IR: Intermediate representation
- Frontend: Legal code → IR
  - (Typ.) Target-independent
- Middle-end: IR → Optimized IR
  - (Typ.) Target-independent
- Backend: IR → Target code
  - Target-dependent



Adapted from: A. Appel: Modern Compiler Implementation in C.

Structure of a compiler: Front-end + Middle-end

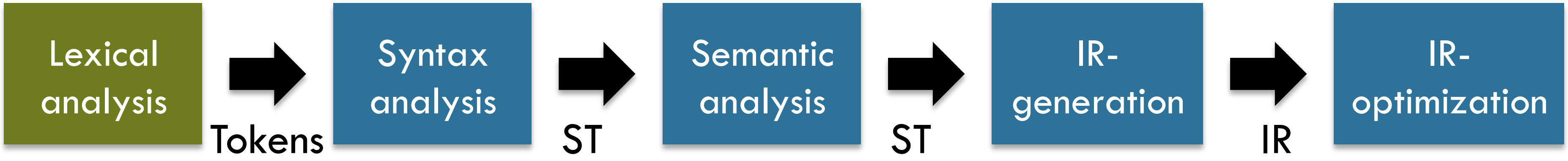


- Lexical (scanner): Maps a character stream into words (tokens)
- Syntax (parser):
  - Recognizes “sentences of tokens” according to a grammar
  - Produces a representation of the application: **Syntax Tree** (ST)
- Semantic analysis: Adds information and checks – types, declarations, …
- IR-generation: Abstract representation of the program, amenable for code generation (backend) – A (abstract) syntax tree is a form of IR
- IR-optimization: Simplification & improvements (e.g., remove redundancies)

Structure of a compiler: Backend

- Code selection: Decide which instructions should implement the IR
- Register allocation: Decide in which register to place variables
- Scheduling: Decide when to execute the instructions (e.g., ordering in assembly program) & ensure conformance with interfaces and constraints

Lexical analysis – Example




```
while (y < z)
{
  int x = a + b;
  y += x;
}
```

```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int T_Identifier x
```

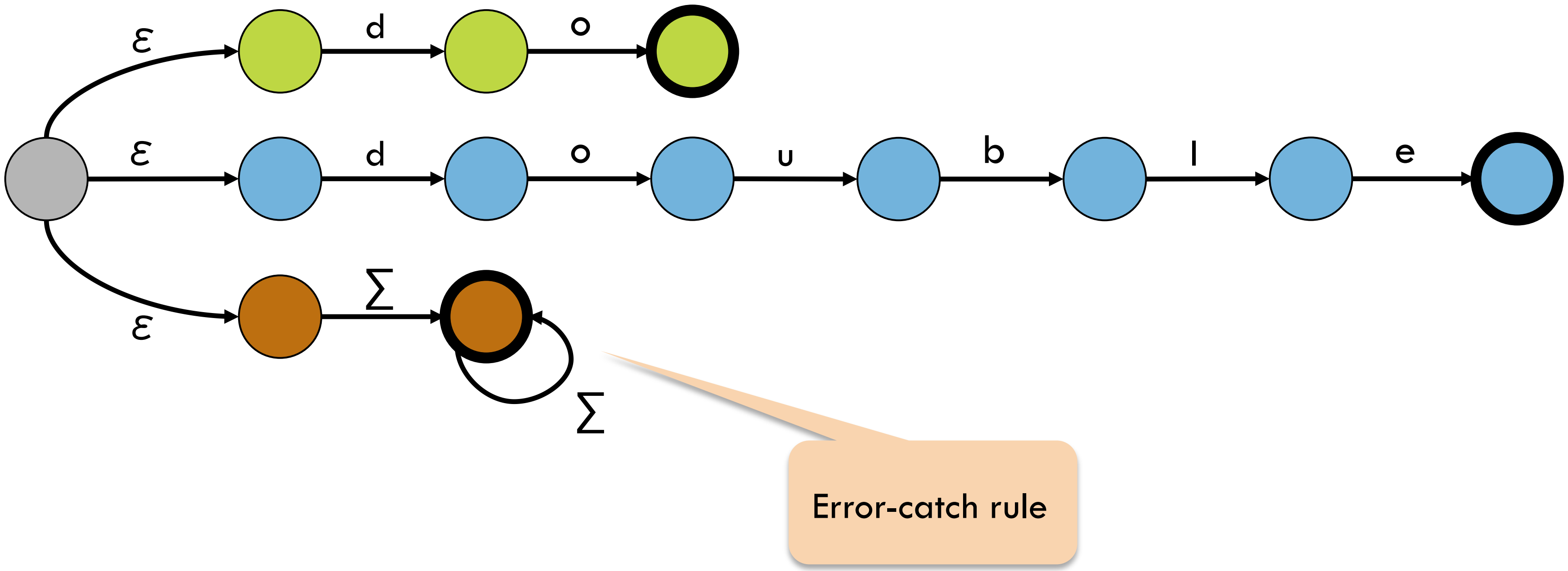
```
T_Assign T_Identifier a
T_Plus T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

Adapted from: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

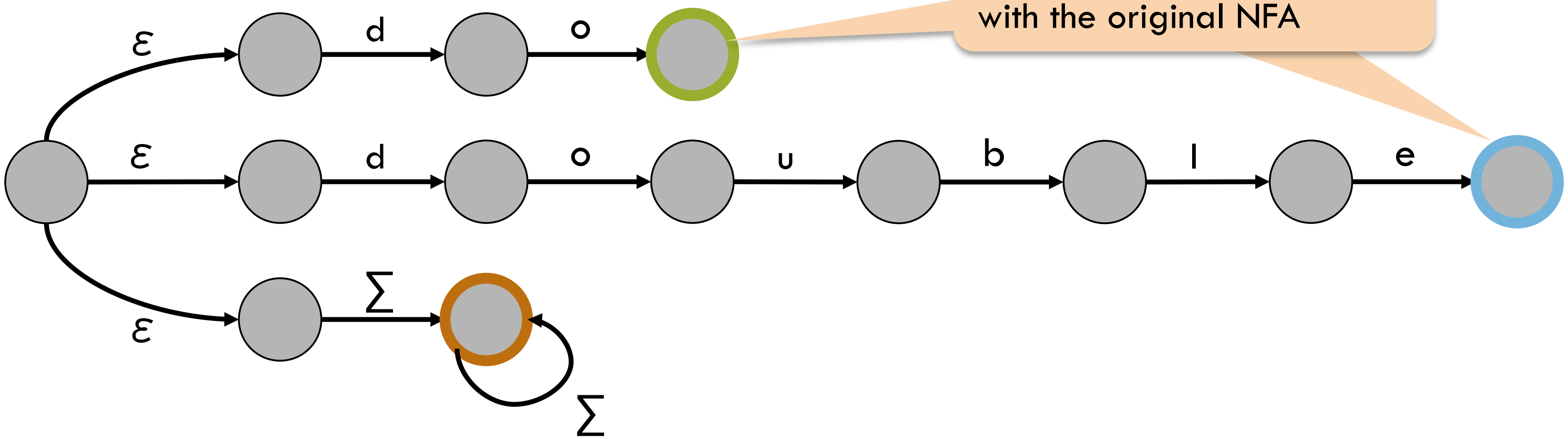
Lexical analysis – The how

- Use **regular expressions** to represent language elements (**tokens**)
  - Identifiers, integer/real constants, keywords (for, do, while, …)
- Examples
  - Binary numbers: (1|0)*
  - Identifiers: [a-zA-Z] [a-zA-Z0-9]*
- Principle of operation
  - Describe all tokens with **regexp**
  - Create **finite state automaton** (FSA) for each regexp
  - Run all FSAs in parallel while reading the input source code

Operation example



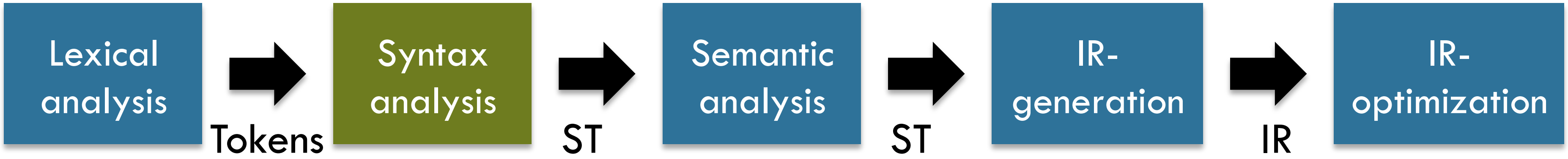
Operation example



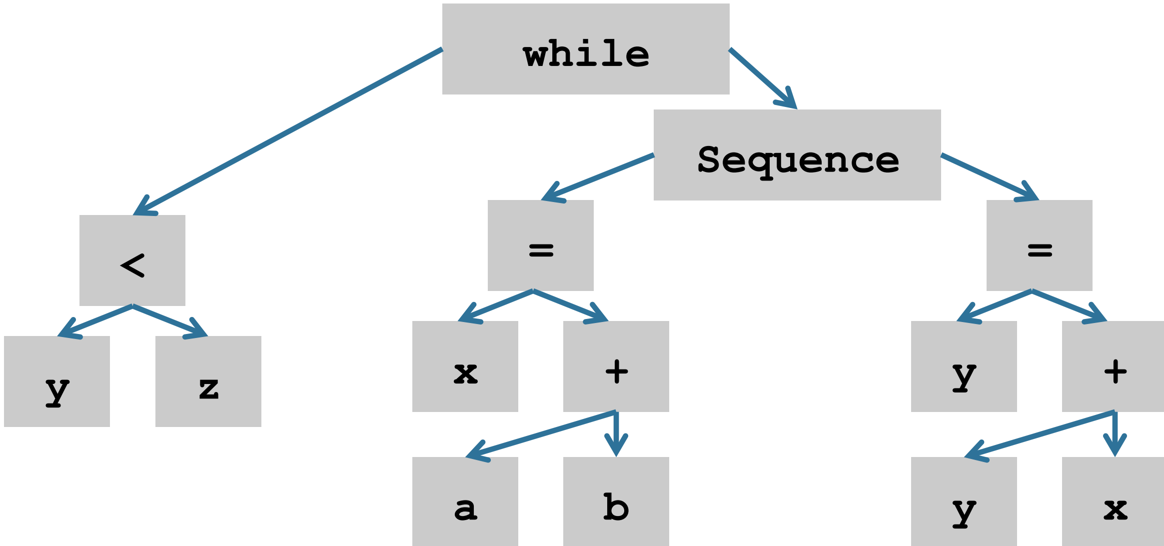
Annotate accepting space with the original NFA

There are tools that automate the process regexp → automata, example: lex/flex

Syntax analysis – Example




```
while (y < z)
{
  int x = a + b;
  y += x;
}
```

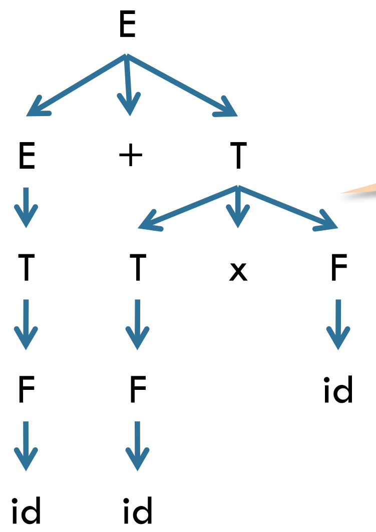


Adapted from: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

Syntax analysis – The how

- Structure of programming languages cannot be captured by regexp
- Instead: **Context Free Grammars** (sloppy: a set of rules to produce programs)
- Example

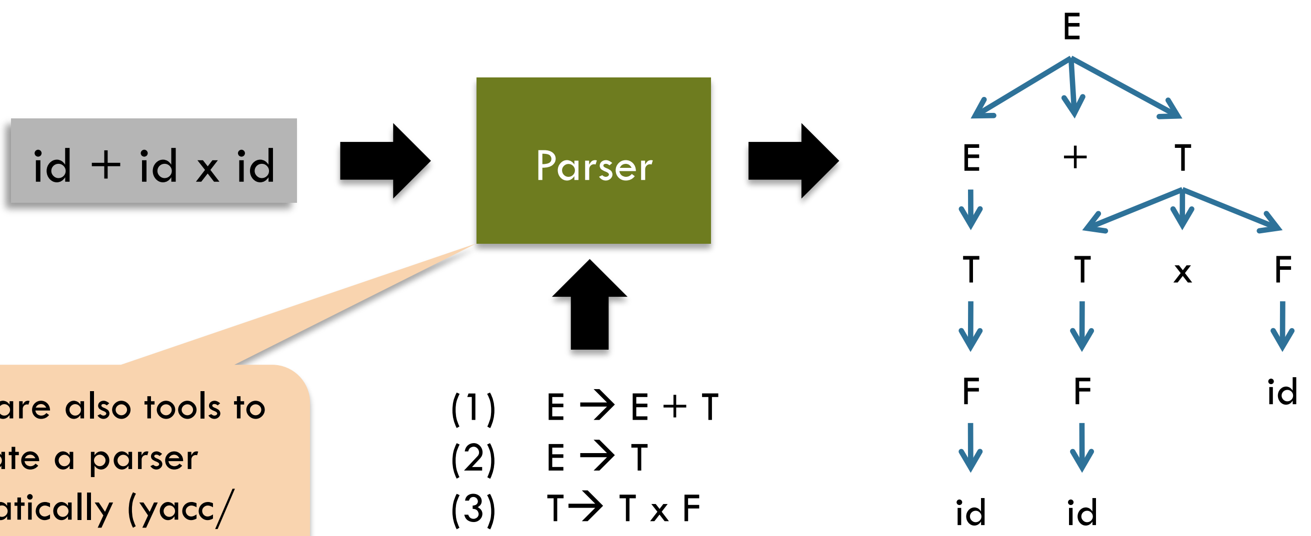
(1) E → E + T
(2) E → T
(3) T→ T x F
(4) T → F
(5) F → id



This sequence of rules produced the program:
id + id x id

Syntax analysis: Parsing

- We are interested in the opposite task
- Parsing: Given a grammar and a program, find the **syntax tree**



There are also tools to generate a parser automatically (yacc/ bison, ANTLR)

(1) E → E + T
(2) E → T
(3) T → T x F
(4) T → F
(5) F → id

Or error!

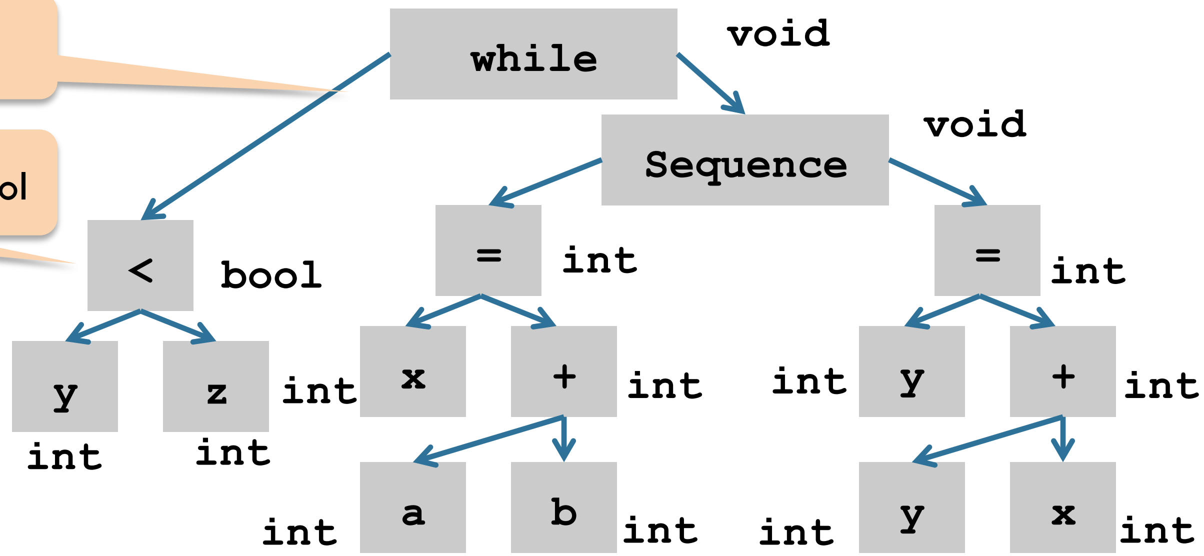
Semantic analysis – Example

Lexical analysis ➔ Tokens ➔ Syntax analysis ➔ ST ➔ **Semantic analysis** ➔ ST ➔ IR-generation ➔ IR ➔ IR-optimization

ST: Syntax Tree (= parse tree)

Semantic: int < int ➔ produces a bool

```
while (y < z)
{
  int x = a + b;
  y += x;
}
```



Why semantic analysis?

```
foo(int a,int b,int c,int d)
{...}
bar()
{
  int f[3],g[0], h, i, j, k;
  char *p;
  foo(h,i,"ab",j, k);
  k = f * i + j;
  h = g[17];
  printf("<%s,%s>.\n", p,q);
  p = 10;
}
```

To generate code, the compiler must understand meaning

- Wrong number of arguments and type
- Wrong dimension for f
- Accessing g[17], but declared g[0]
- q unknown
- p is "char*"

Adapted from: Keith D. Cooper, Ken Kennedy & Linda Torczon

Semantic analysis – The how

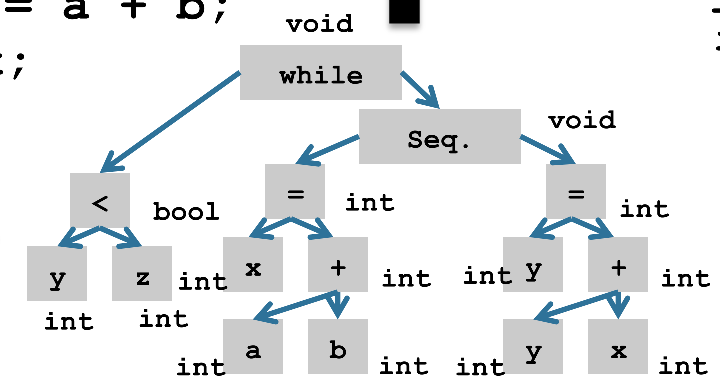
- Different approaches
  - Formal: Attribute grammars
  - Programmatic: Visitor pattern on the syntax tree (used by LLVM)

- Symbol table: Data structure that stores object names and their attributes
  - Populated by walking the syntax tree
  - At any point of the tree
    - The compiler knows which variables, of which type, are available
    - Possible to check for correctness

IR generation & optimization – Example

Lexical analysis → Tokens → Syntax analysis → ST → Semantic analysis → ST → IR-generation → IR → IR-optimization

```
while (y < z)
{
  int x = a + b;
  y += x;
}
```



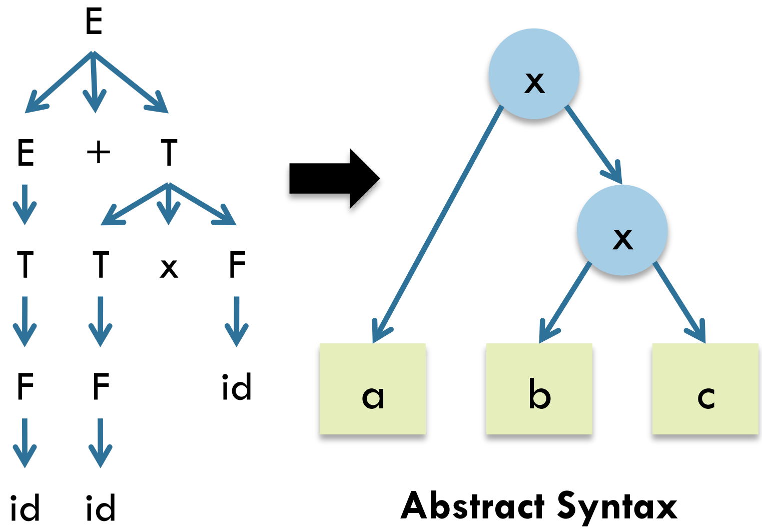

```
Loop: x = a + b
      y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

Abstract representation of the program, closer to the target

```
      x = a + b
Loop: y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

IR generation & optimization – The how

- Walk the syntax tree to generate different implementations
  - The syntax tree is itself a for of IR
  - Use simple "code generation"
  - Optimize afterwards



Abstract Syntax Tree (AST)

LLVM IR: Three-address code (TAC)

- Example: LLVM Framework

```
int main()
{
  int a, b, c;
  a = 2; b = 3;c = 5;
  c += (a*b) >> a;
  c += foo(a);
  return 0;
}
```

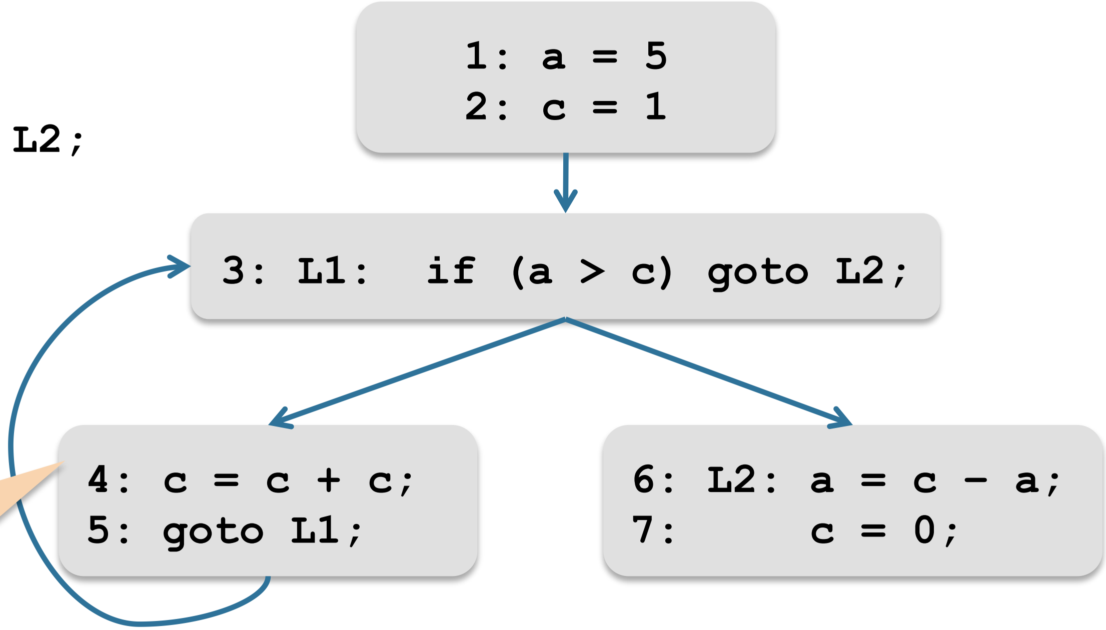
```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  store i32 0, i32* %1
  store i32 2, i32* %a, align 4
  store i32 3, i32* %b, align 4
  store i32 5, i32* %c, align 4
  %2 = load i32* %a, align 4
  %3 = load i32* %b, align 4
  %4 = mul nsw i32 %2, %3
  %5 = load i32* %a, align 4
  %6 = ashr i32 %4, %5
  %7 = load i32* %c, align 4
  %8 = add nsw i32 %7, %6
  store i32 %8, i32* %c, align 4
  %9 = load i32* %a, align 4
  %10 = call i32 @foo(i32 %9)
  %11 = load i32* %c, align 4
  %12 = add nsw i32 %11, %10
  store i32 %12, i32* %c, align 4
  ret i32 0}
```

A lot of room for optimization!

Graph representations: control flow graph

- Control flow graph: Represent the branching structure of programs

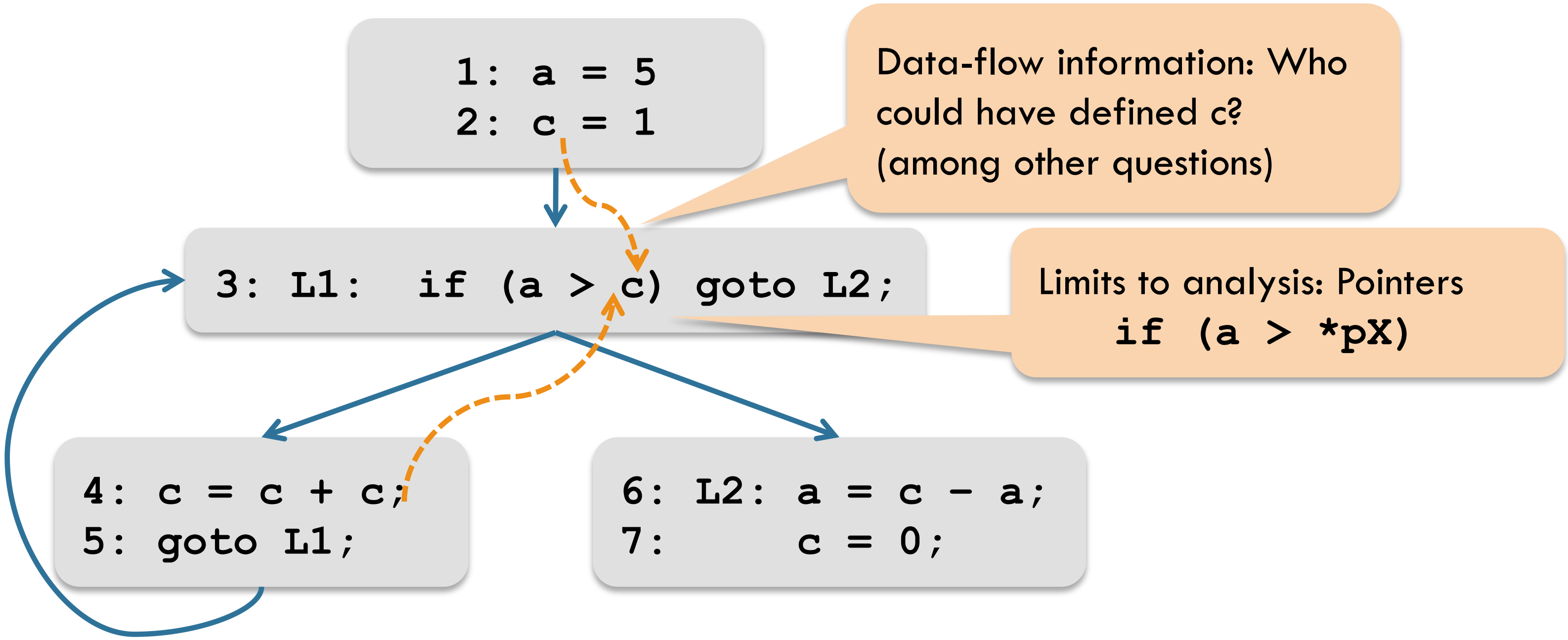
```
1:       a = 5;
2:       c = 1;
3: L1:   if (a > c) goto L2;
4:       c = c + c;
5:       goto L1;
6: L2:   a = c - a;
7:       c = 0;
```



Basic-blocks: sequence of statements w/o branching in between

Graph representations: data-flow information

- The compiler has to know where the data is coming from for optimizations



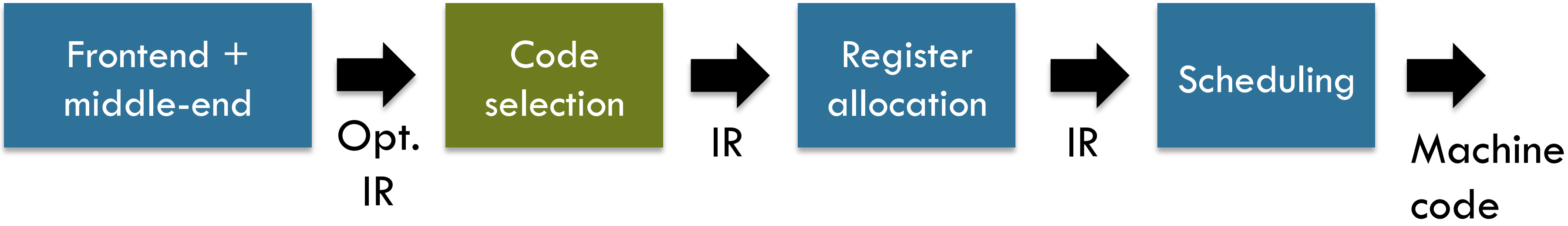
Backend – Example


```
while (y < z)
{
  int x = a + b;
  y += x;
}
```

```
      x = a + b
Loop: y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

```
      ADD R1, R2, R3
Loop: ADD R4, R1, R4
      SLT R6, R4, R5
      BEQ R6, loop
```

Code selection – The how

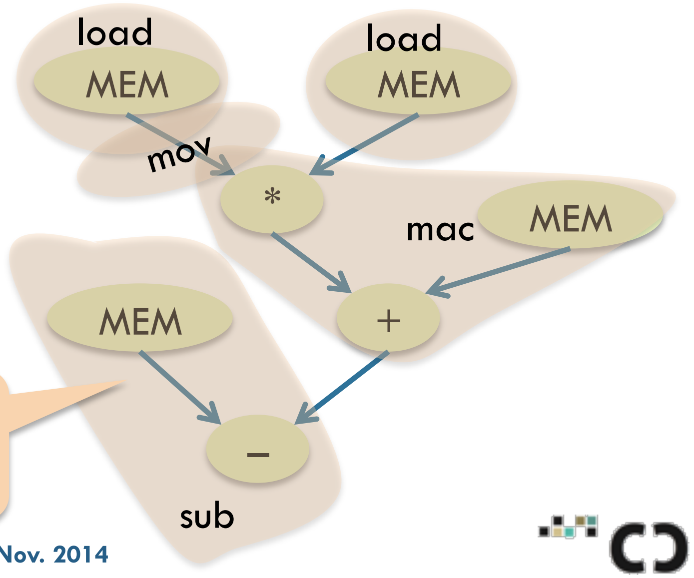


- Principle: pattern matching in some form of IR (e.g., ASTs)

```
x = a - ((b * c) + d)
```

```
LDR R1, $b
MOV R1, R2
LDR R1, $c
MAC R1, R1, R2, $d
SUB R1, $a, R1
STR R1, $x
```

Based on a cost-model of the ISA

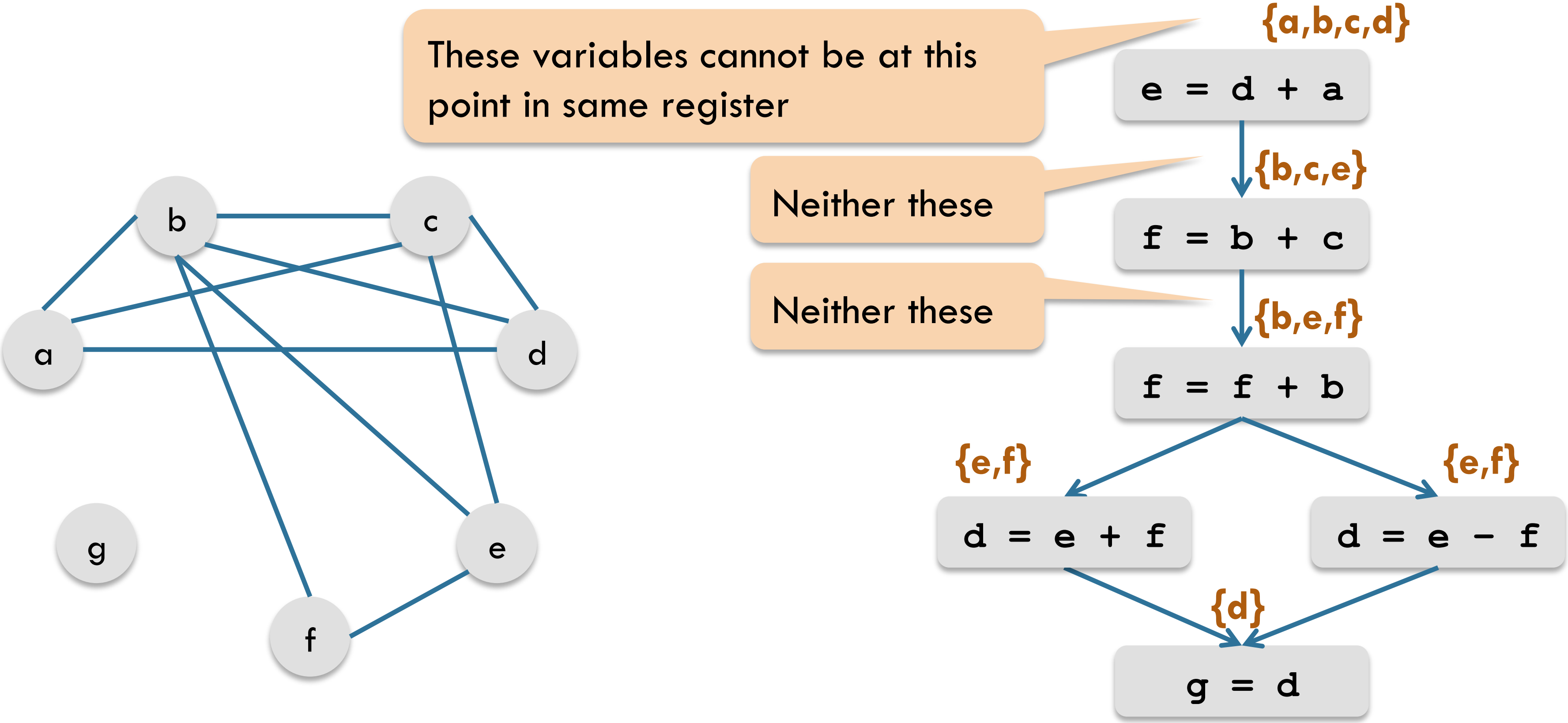


Register allocation – The how

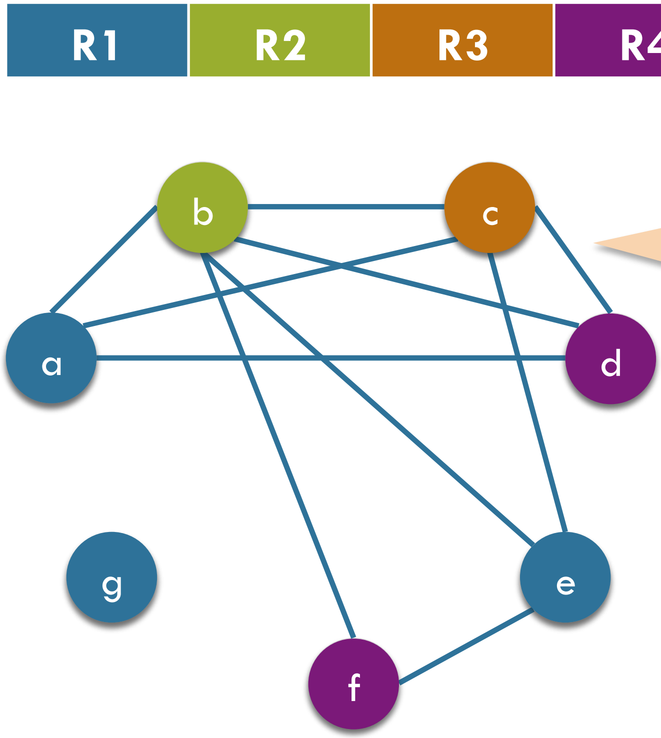
Frontend + middle-end → Opt. IR → Code selection → IR → Register allocation → IR → Scheduling → Machine code

- Reduce the amount of spill code (e.g., storing variables in the stack)
- Find out life-ranges of variables (with a form of data flow analysis)
- Apply graph coloring

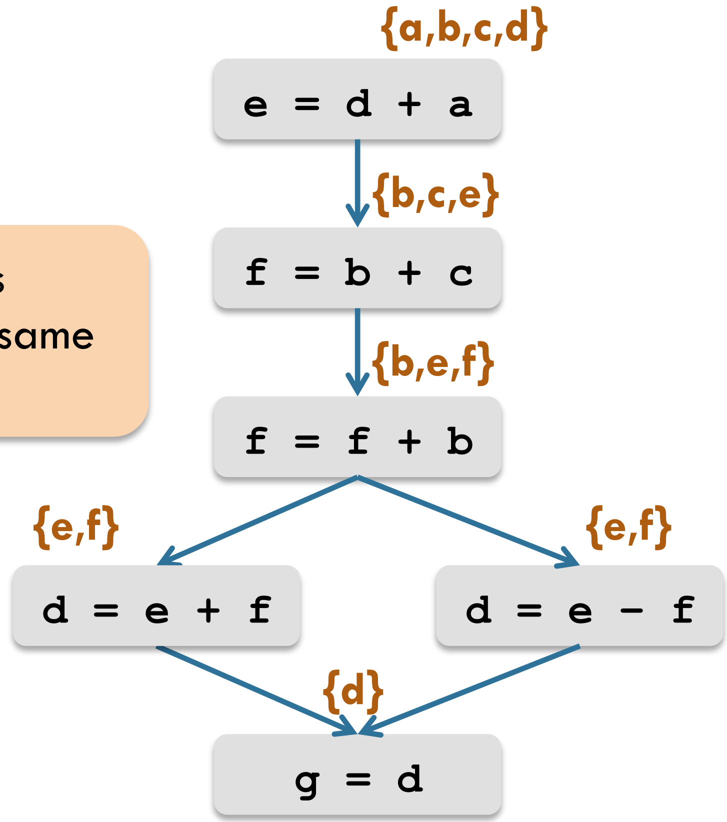
Register allocation – graph coloring



Register allocation – graph coloring (2)



Connected nodes cannot be in the same register



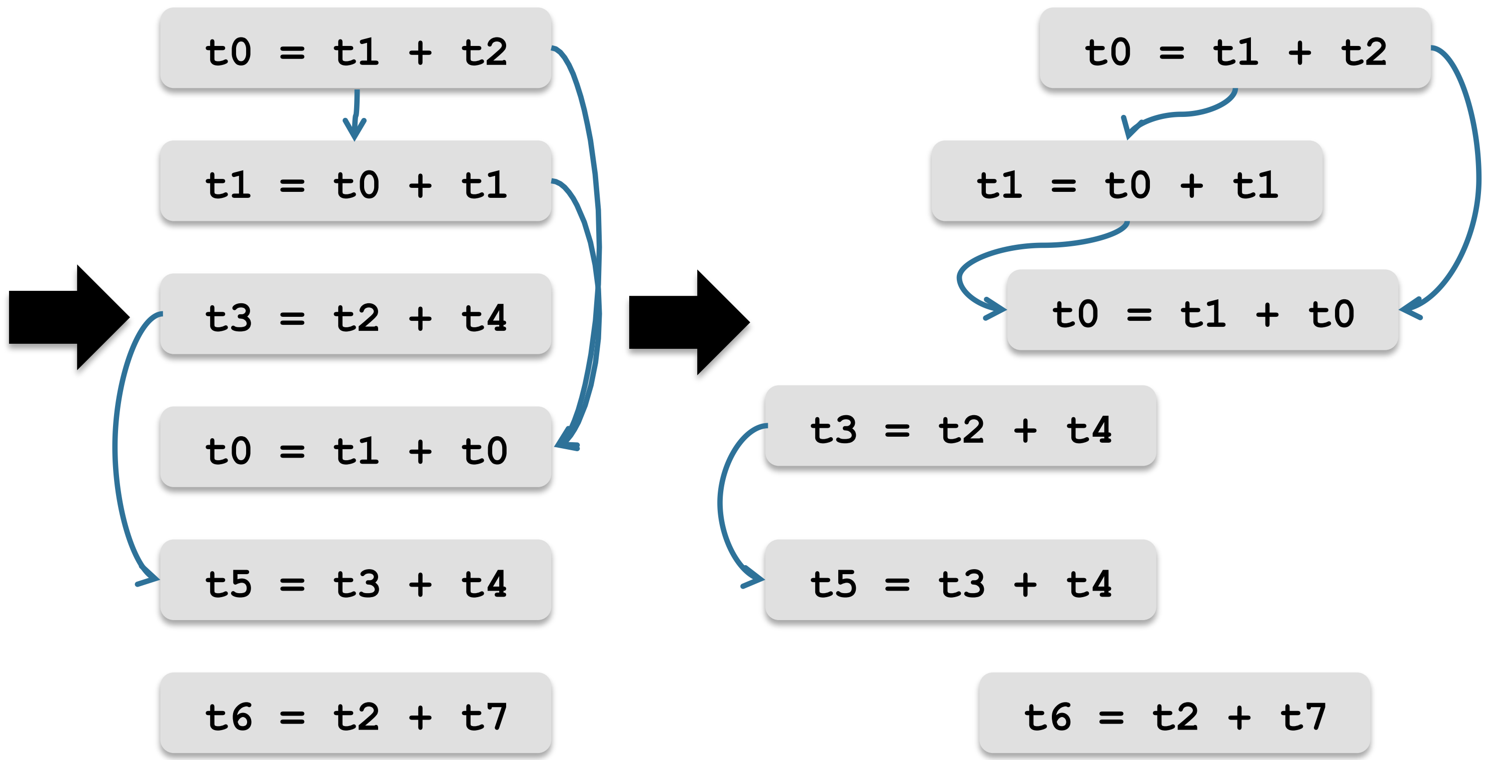
Scheduling – The how

Frontend + middle-end → Opt. IR → Code selection → IR → Register allocation → IR → **Scheduling** → Machine code

- Find an order of instructions that minimizes the execution time
- Has to respect data dependencies (a form of data analysis)

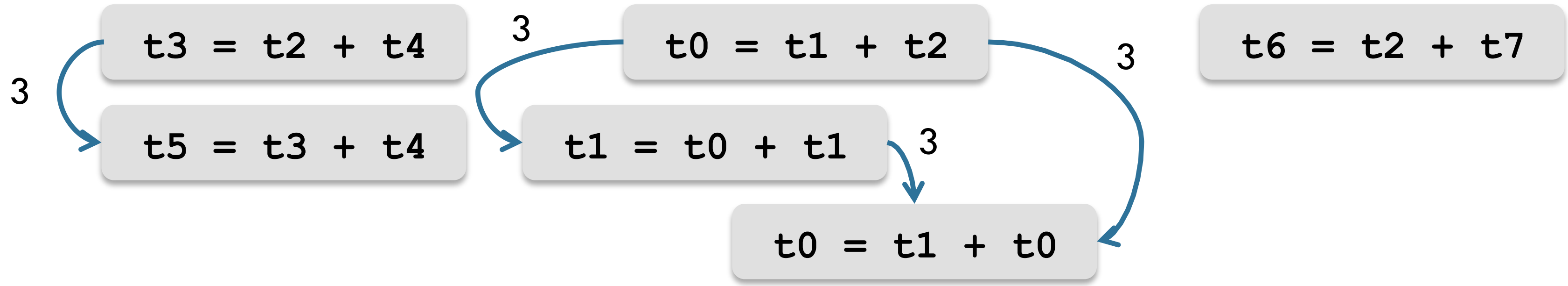
Scheduling – dependency graph (inside basic block)

```
t0 = t1 + t2
t1 = t0 + t1
t3 = t2 + t4
t0 = t1 + t2
t5 = t3 + t4
t6 = t2 + t7
```



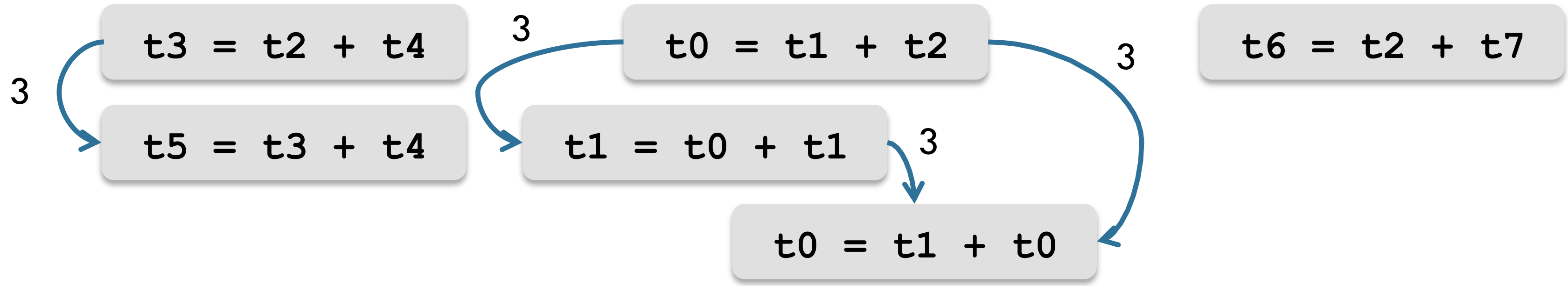
Adapted from: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

Schedules on a RISC processor: option 1



Option 1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
t3 = t2 + t4	IF	ID	EX	MM	WB											
t5 = t3 + t4				IF	ID	EX	MM	WB								
t0 = t1 + t2					IF	ID	EX	MM	WB							
t1 = t0 + t1								IF	ID	EX	MM	WB				
t0 = t1 + t0											IF	ID	EX	MM	WB	
t6 = t2 + t7												IF	ID	EX	MM	WB

Schedules on a RISC processor: option 2



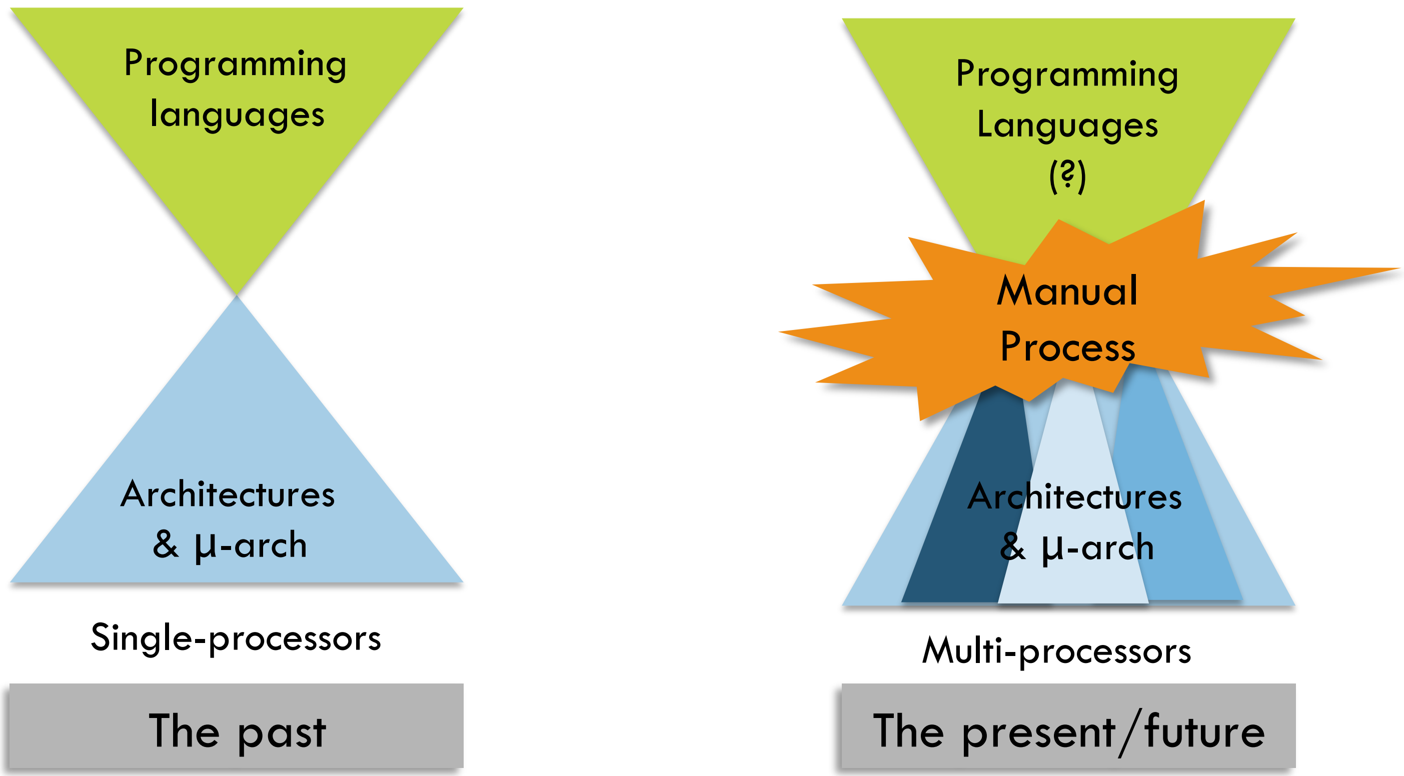
Option 2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
t0 = t1 + t2	IF	ID	EX	MM	WB											
t3 = t2 + t4		IF	ID	EX	MM	WB										
t6 = t2 + t7			IF	ID	EX	MM	WB									
t1 = t0 + t1				IF	ID	EX	MM	WB								
t5 = t3 + t4					IF	ID	EX	MM	WB							
t0 = t1 + t0							IF	ID	EX	MM	WB					

Saved cycles

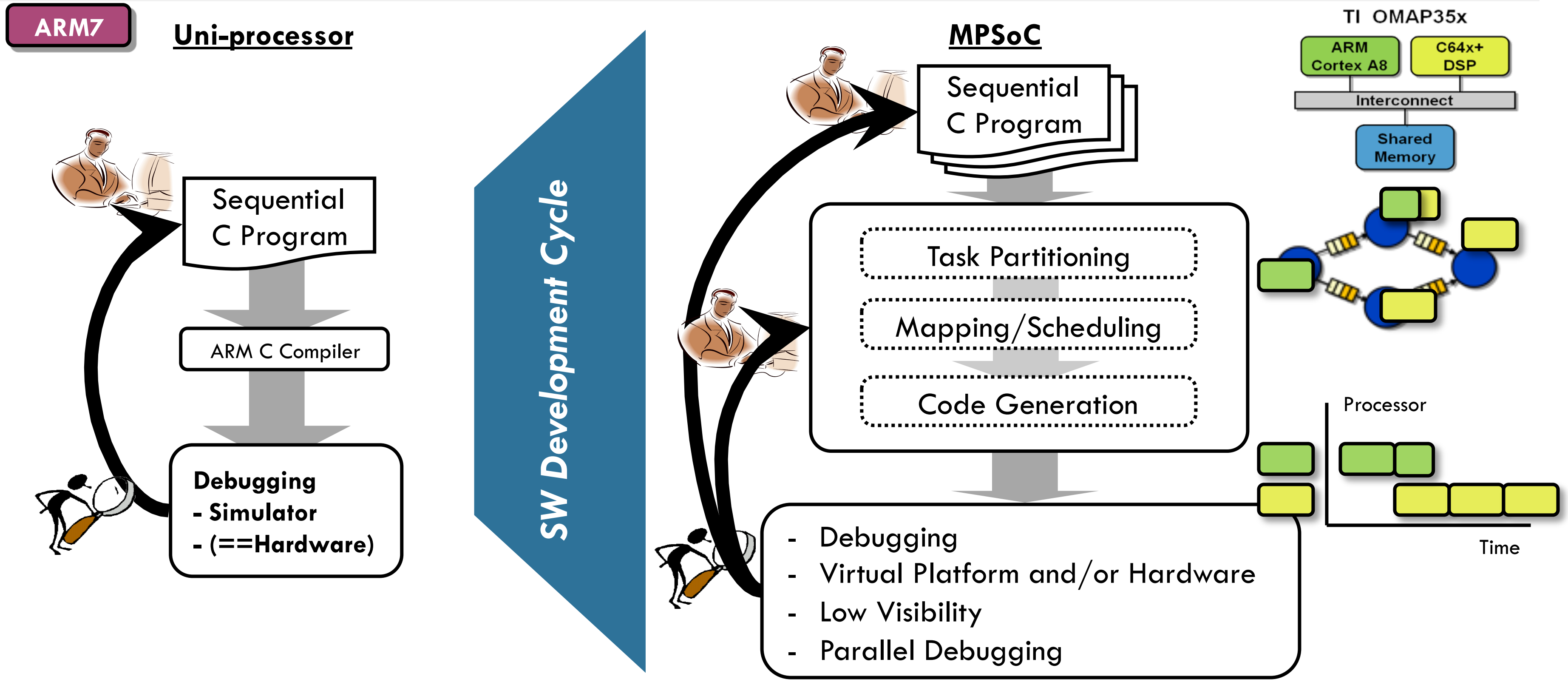
3. Insight multi-core compilers

- Parallelizing sequential codes
- Parallel dataflow models

Recall: challenges in multi-core compilation



Uni-processor vs Multi-processor



Multi-core compilers

- Deal with similar problems than classical compilers
  - Parse and understand high-level parallel language constructs
  - Search for parallelism, but at a higher level of abstraction (higher than ILP)
  - Requires a model of the target architecture, but at coarser level
  - Allocation and scheduling of data to memories and tasks to processors
  - Code generation via source-to-source compilation

On parallel programming models

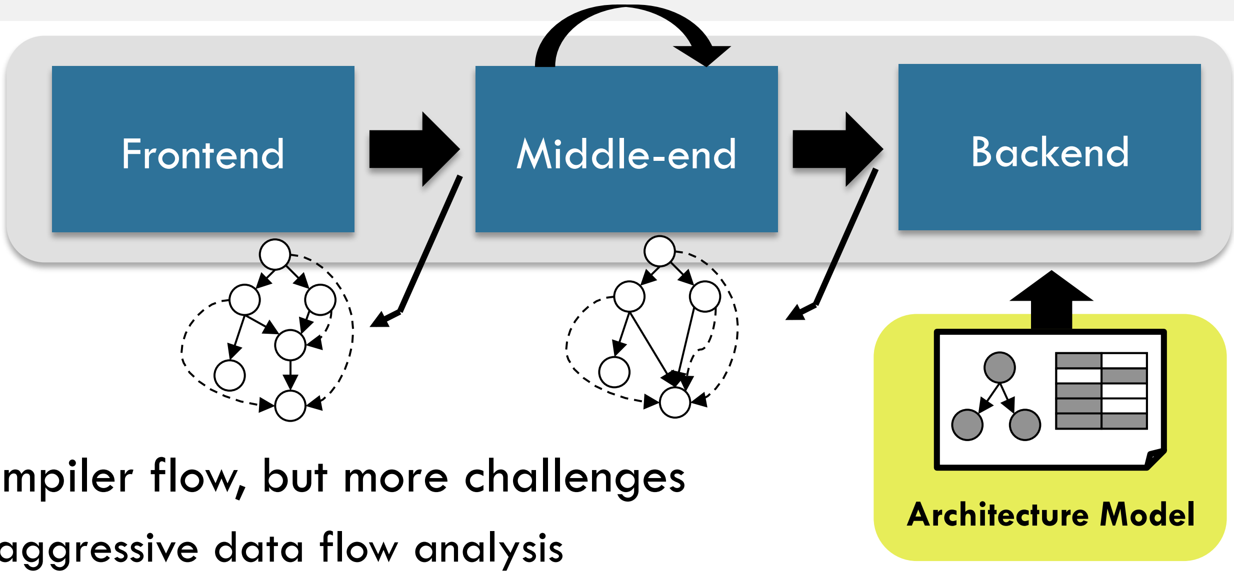
- There are many, really many: With different impacts on compilers
- Sequential programming models: C/C++, Matlab, …
- Parallel programming models
  - Shared memory: pthreads, OpenMP, Intel TBB, Cilk, …
  - Distributed memory: MPI, Charm++, …

- In this presentation
  - Extracting coarse-grained parallelism from C code
  - Parallel dataflow models

3. Insight multi-core compilers

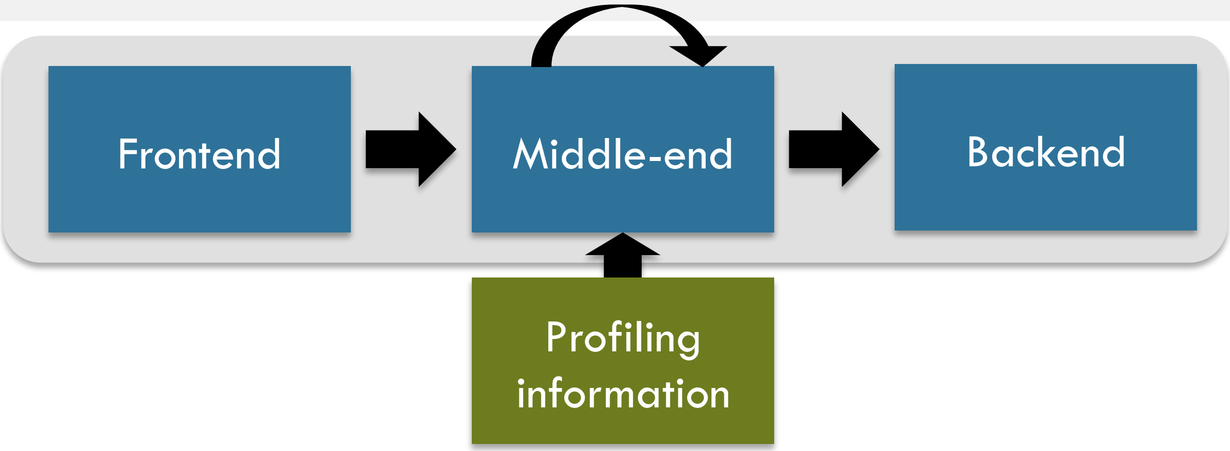
- Parallelizing sequential codes
- Parallel dataflow models

Principle of operation



- Similar compiler flow, but more challenges
  - More aggressive data flow analysis
  - More aggressive program transformations
  - Different granularity (basic-blocks?, functions?)
  - Focus on coarse-grained parallelism patterns
  - Whole program analysis

Data-flow analysis



- **Dynamic** data flow analysis via execution traces
  - More exact: Find exact portions of memory being read/written
  - Not sound: Cannot completely rely on dynamic information

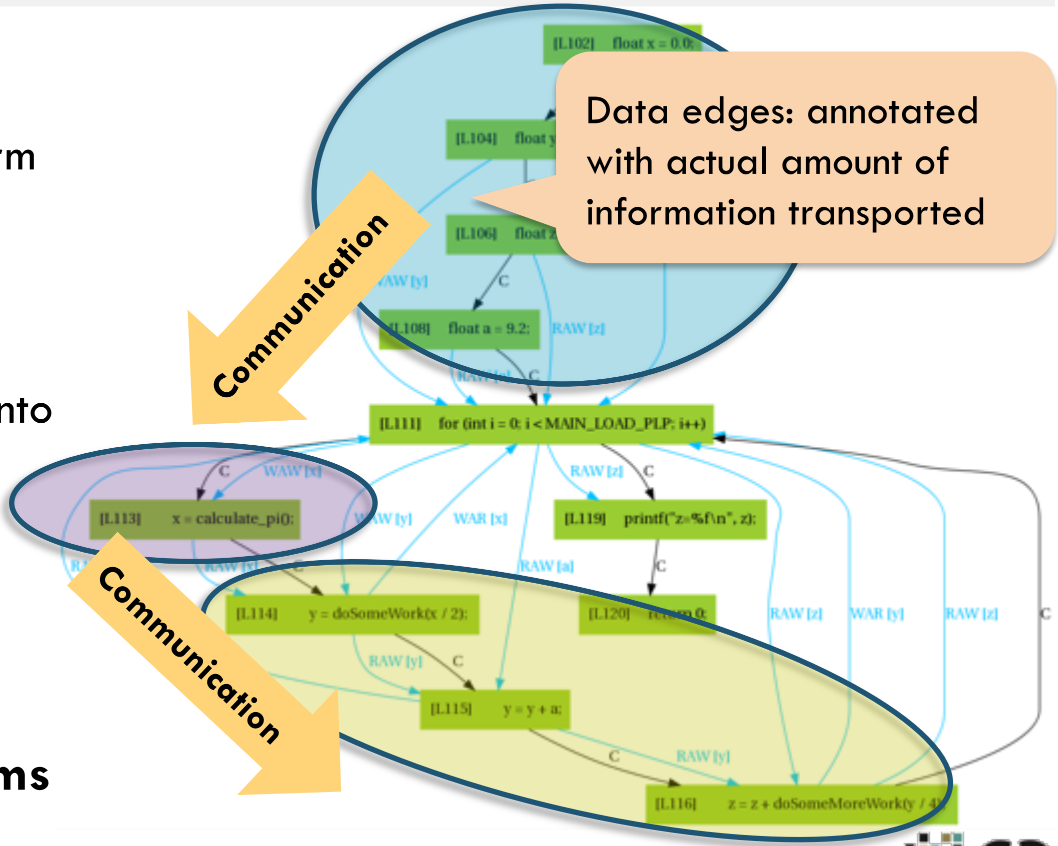
Dynamic data-flow analysis

```
int main(void)
{

    float x = 0.0;

    float y = 0.0;

    float z = 0.0;

    float a = 9.2;


    for (int i = 0; i < MAIN_LOAD_PLP; i++)
    {
        x = calculate_pi();
        y = doSomeWork(x / 2);
        y = y + a;
        z = z + doSomeMoreWork(y / 4);
    }

    printf("z=%f\n", z);
    return 0;
}
```

On granularity

- Granularity for analysis
  - Depends on the target platform
  - Requires whole-program information: Costs of functions called
  - Need to take communication into account
  - Is not given by traditional compiler boundaries: basic-blocks or functions

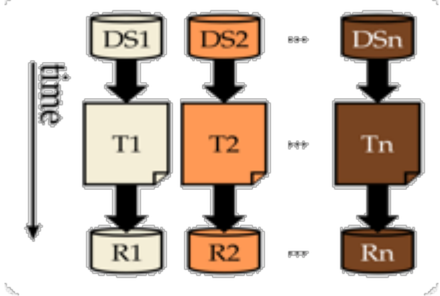
➔ **Use graph clustering algorithms**



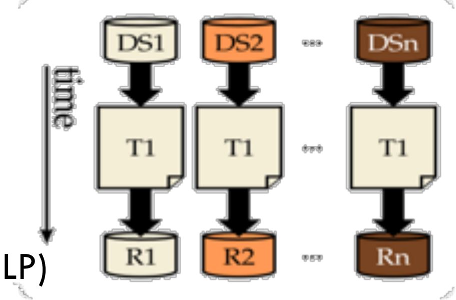
Course-grained parallelism patterns

- Search for known parallelism patterns
  - Task-level parallelism
  - Data-level parallelism
  - Pipeline-level parallelism
  - Others: Reduction, commutative operations, …

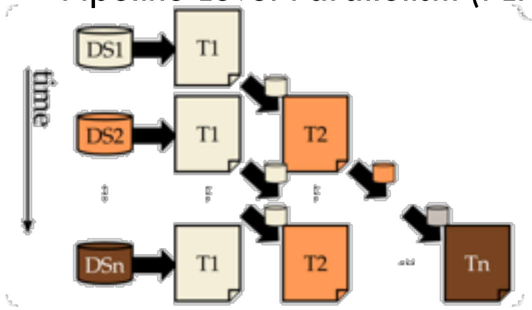
Task Level Parallelism (TLP)



Data Level Parallelism (DLP)



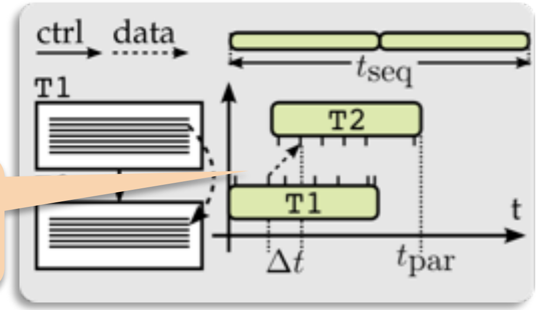
Pipeline Level Parallelism (PLP)

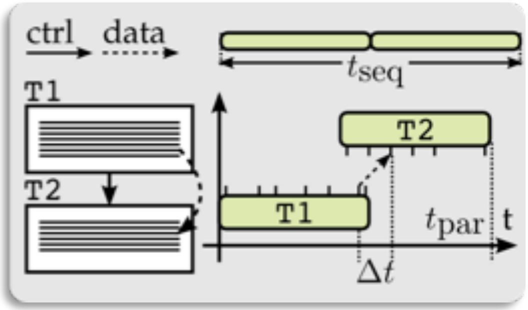


Judging parallelism patterns

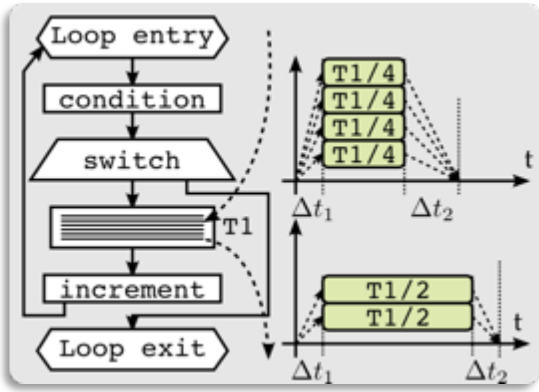
- TLP examples

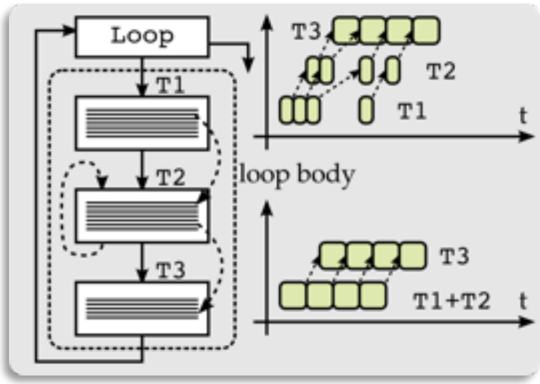
Need parallel performance estimation





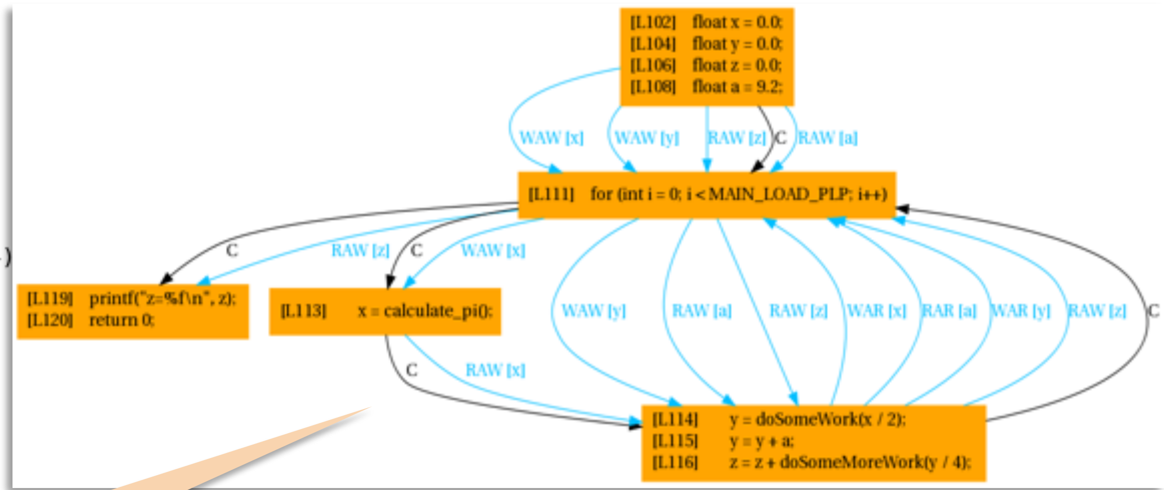
- DLP and PLP





PLP example

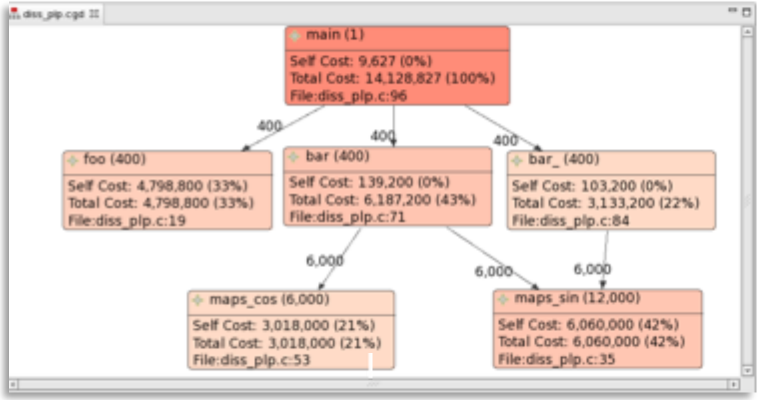
```c
int main(void)
{

    float x = 0.0;

    float y = 0.0;

    float z = 0.0;

    float a = 9.2;


    for (int i = 0; i < MAIN_LOAD_PLP; i++)
    {
        x = calculate_pi();
        y = doSomeWork(x / 2);
        y = y + a;
        z = z + doSomeMoreWork(y / 4);
    }

    printf("z=%f\n", z);
    return 0;
}
```



A two-stage pipeline

Whole program analysis

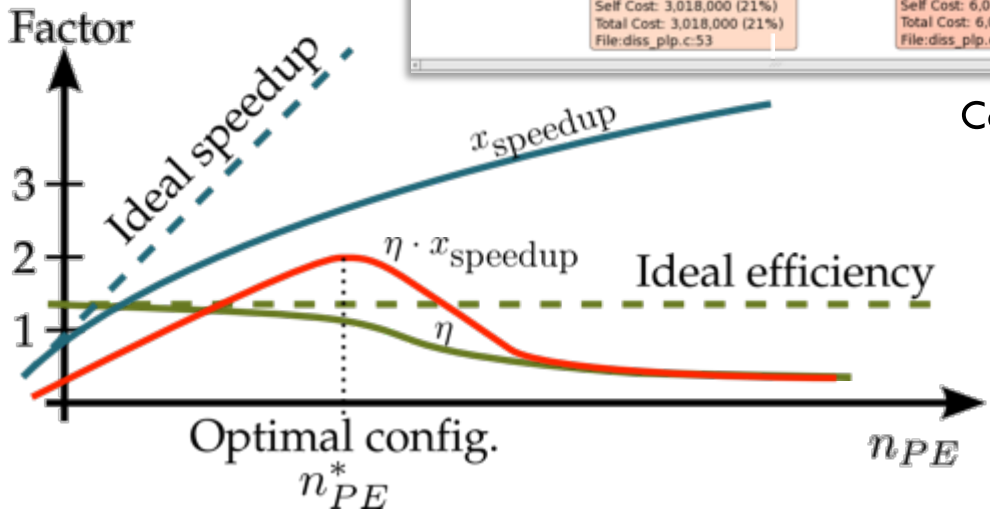
- Global: Discard irrelevant partitions & Fix parameters (e.g., pipe stages)
- Metrics: Efficiency & speedup



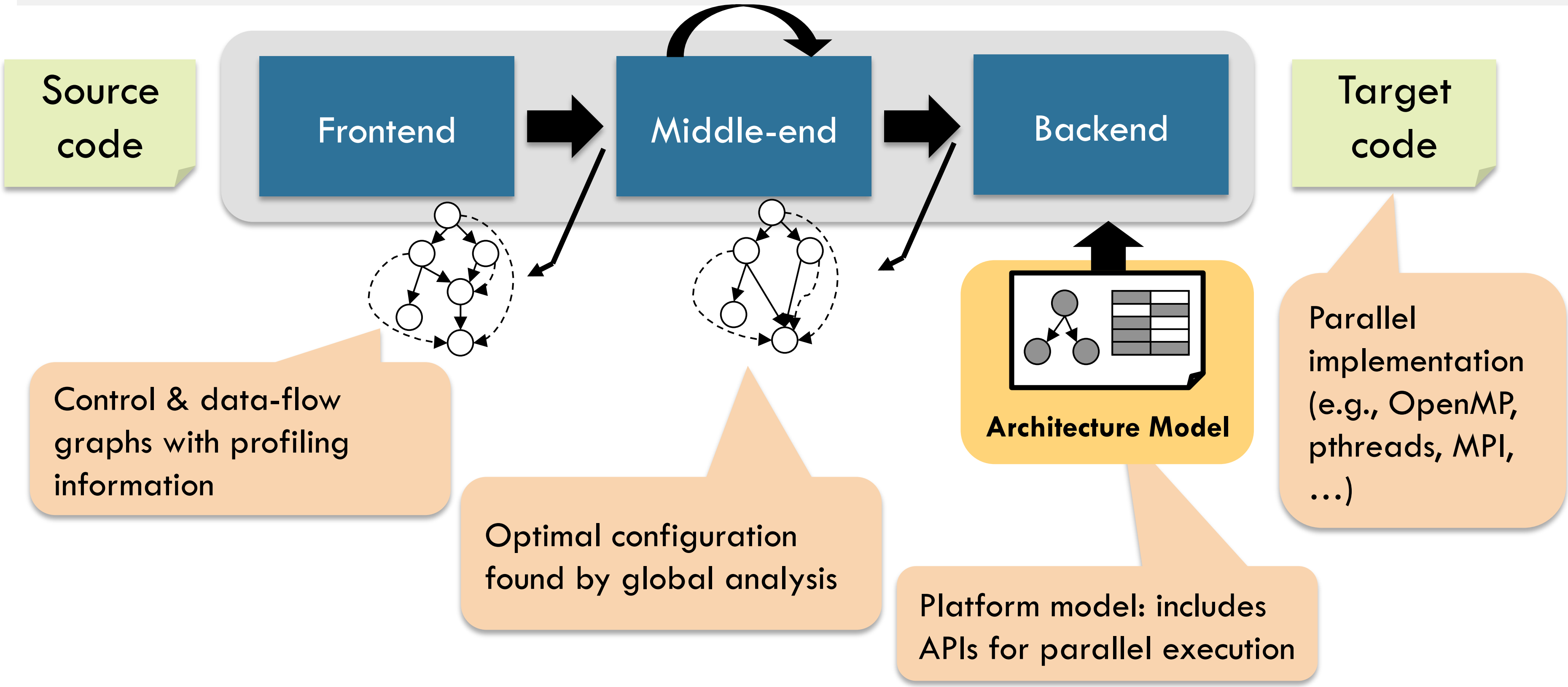
Call graph

$$\eta = \frac{t_{\text{seq}}}{t_{\text{par}} \cdot n_{\text{PE}}}$$

$$x_{\text{speedup}} = \frac{t_{\text{seq}}}{t_{\text{par}}}$$



Entire flow & code generation



Lessons learned

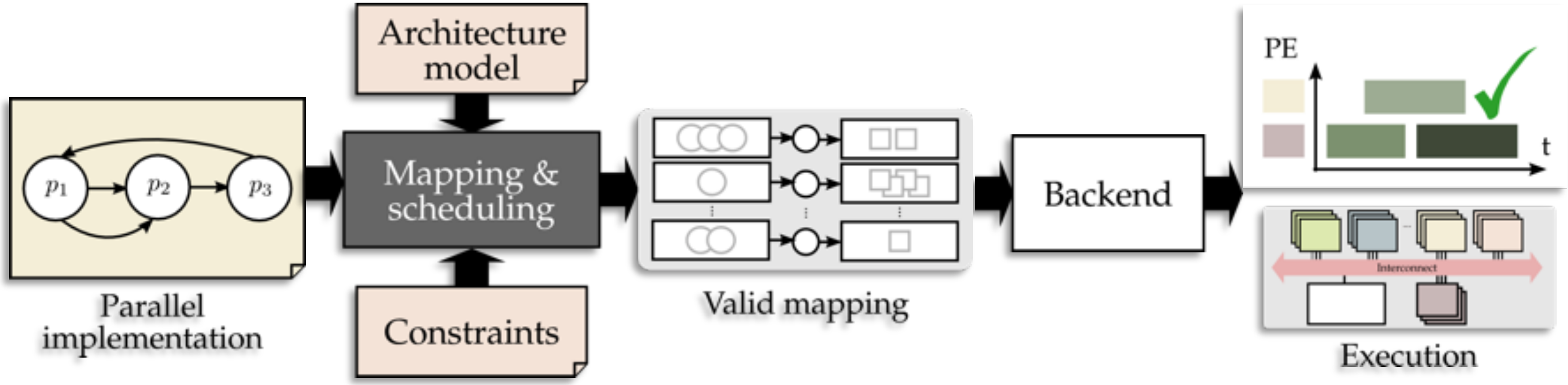
- Automatic parallelism extraction is hard
  - → Semi-automatic approach
- It pays off for some applications depending on coding style
  - Focus on small patterns and do a good job for them
  - Application-specific knowledge is key

- Partitioning data-structures is a must
- High-level performance estimation is important (and difficult)
- Important practical problem but not the solutions for many-cores

3. Insight multi-core compilers

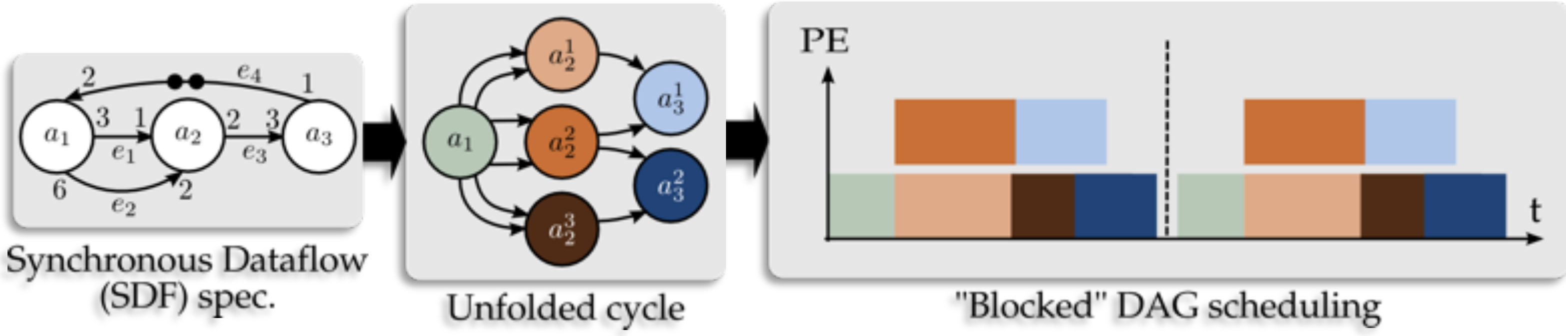
- Parallelizing sequential codes
- Parallel dataflow models

Process networks and dataflow programming



- Kahn Process Networks (KPN) & other flavors of dataflow models
  - A node (process) represents computation
  - An edge (channel) represents communication
- Output: Valid mapping (➔ comply to constraints)
  - Process and channel **mapping**
  - **Buffer sizing:** Memory allocated for communication

Static models: Synchronous Dataflow (SDF)

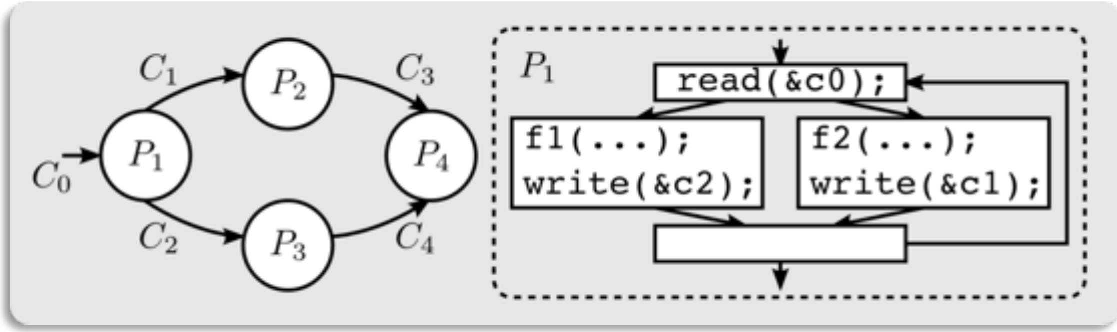


- Fully specified rates, allow more compiler analysis
- Compute **topology matrix**, and solve system of equations
- Solution: **repetition vector** serve to unroll the graph [1 3 2]
- Perform mapping and scheduling on the resulting **directed acyclic graph** (DAG)

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

$$\Gamma \cdot \mathbf{r} = 0$$

Dynamic models: KPNs



- ❑ Channel accesses not visible at the graph
- ➔ Need to look inside the processes

- ❑ Solutions: Use dynamic scheduling
- ❑ Methods: Employ simulations, genetic algorithms or **devise heuristics**

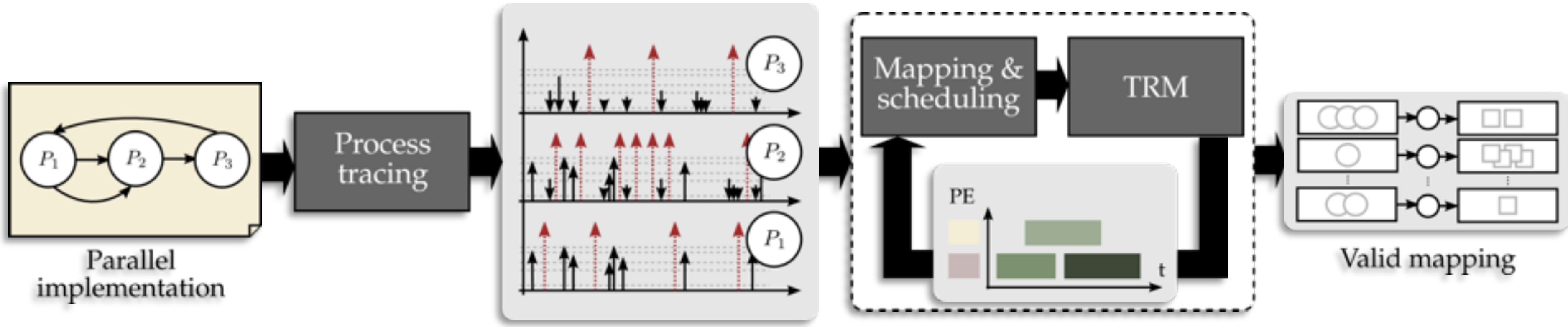
CPN: C for Process Networks

- Channels
- Processes and networks

```
typedef struct { int i; double d; } my_struct_t;
__PNchannel my_struct_t C;
__PNchannel int A = {1, 2, 3}; /* Initialization */
__PNchannel short C[2], D[2], F[2], G[2];
```

```
__PNkpn AudioAmp __PNin(short A[2]) __PNout(short B[2])
                 __PNparam(short boost){
  while (1)
    __PNin(A) __PNout(B) {
      for (int i = 0; i < 2; i++)
        B[i] = A[i]*boost;
    }
}
__PNprocess Amp1 = AudioAmp __PNin(C) __PNout(F) __PNparam(3);
__PNprocess Amp2 = AudioAmp __PNin(D) __PNout(G) __PNparam(10);
```

Trace-based heuristics

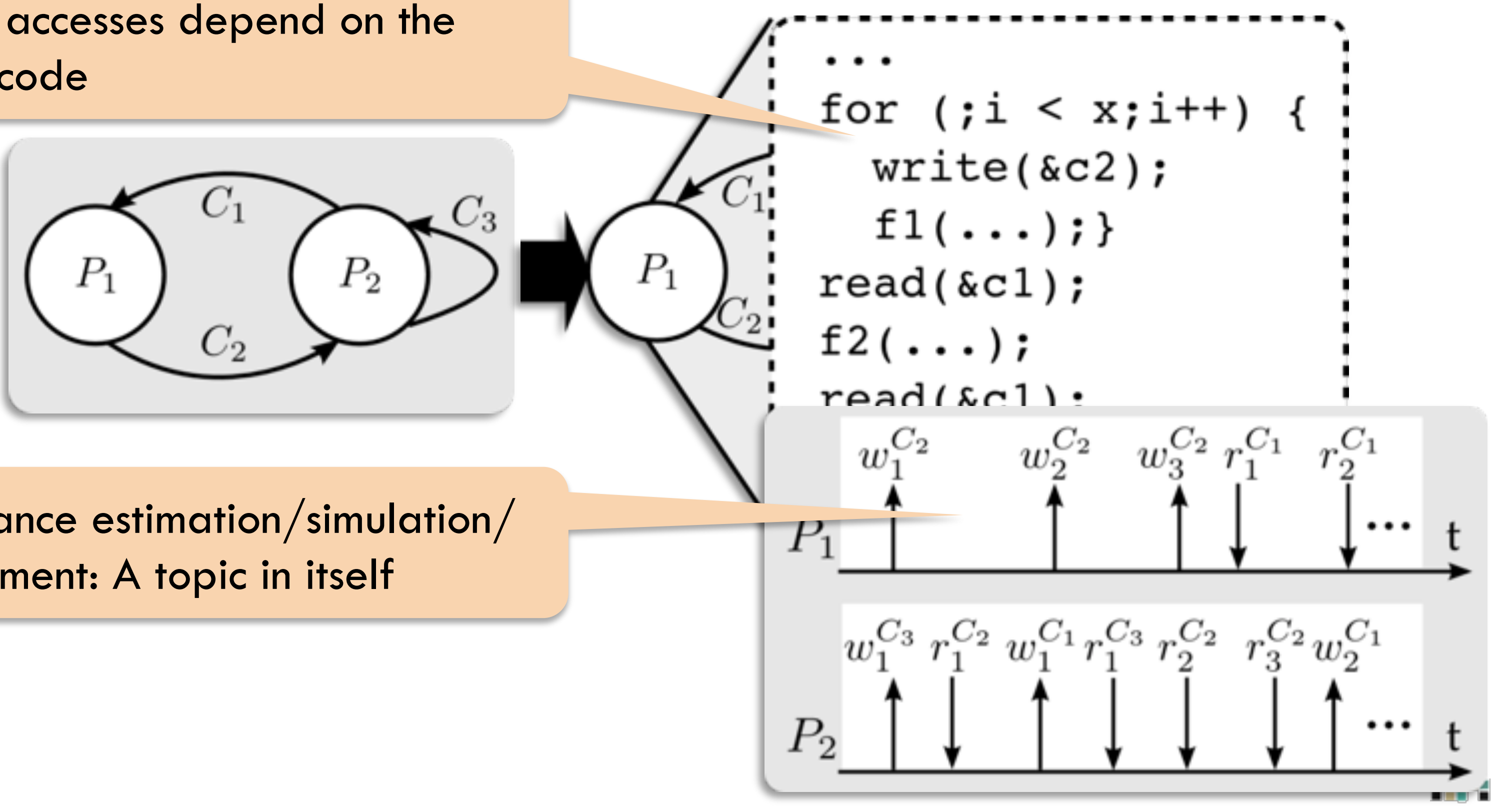


- Process tracing: Understand process interactions
- Mapping
  - Mapping & scheduling: Analyze traces and propose mapping
  - Trace Replay Module (TRM): Evaluate mapping (parallel performance estimation)
  - Iterate: Improve mapping (if required)

Process tracing

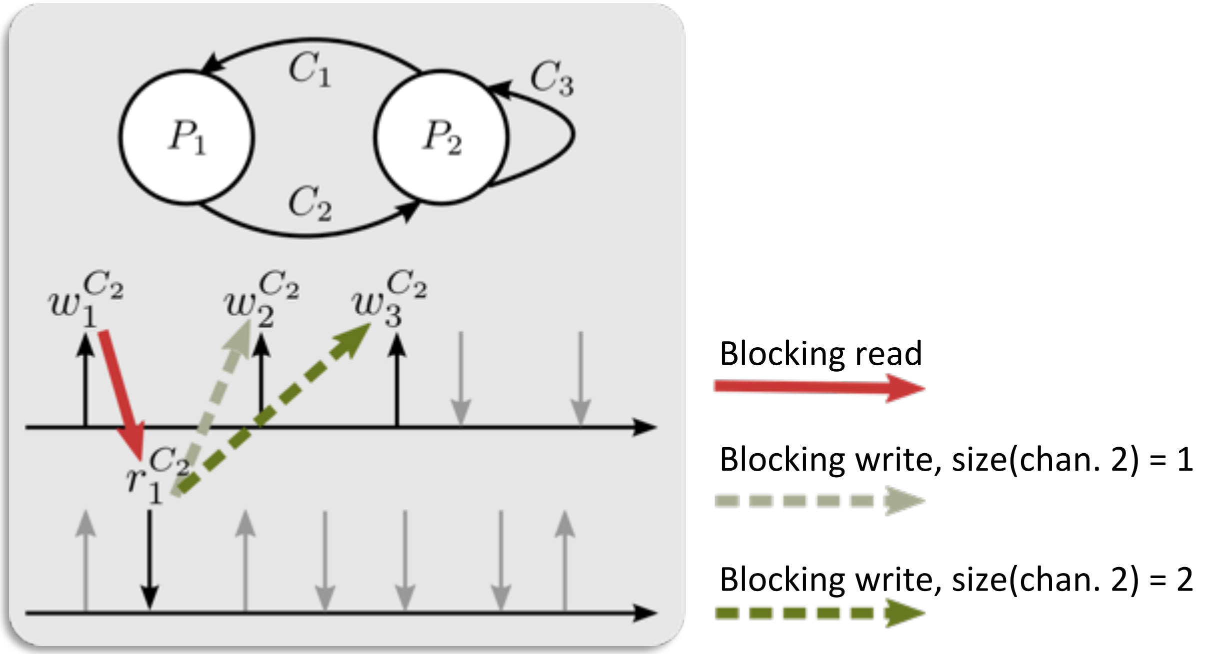
Channel accesses depend on the internal code

Performance estimation/simulation/ measurement: A topic in itself



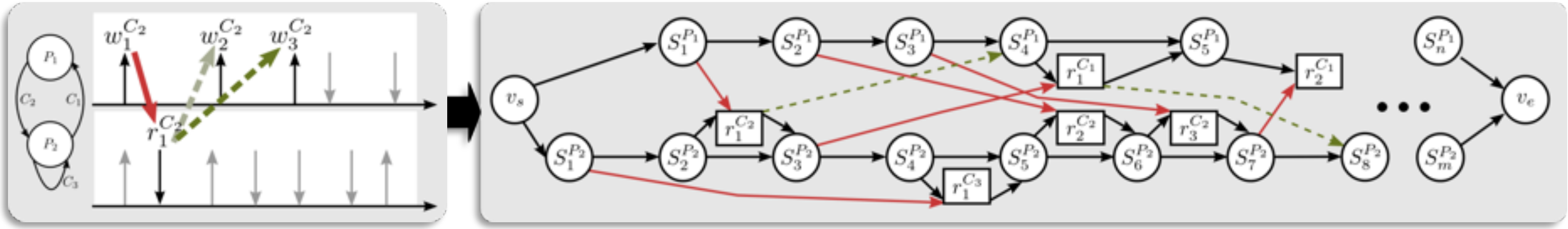
Trace-based algorithms

- Event traces can be represented as large dependence graphs



Trace-based algorithms (2)

- Sample trace graph



size(chan. 2) = 2, size(chan. 1,3) = 1

- Possible to reason about
  - Channel sizes and memory allocation
  - Mapping and scheduling onto heterogeneous processors

Example algorithm: Group-based mapping (GBM)

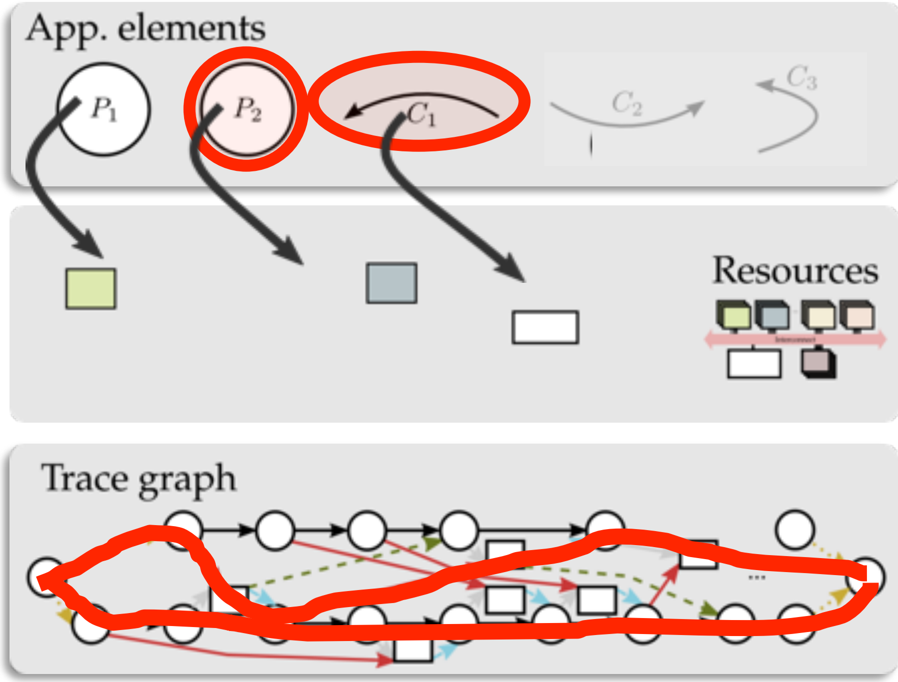
1) Initialize: All to all

2) Select element: Trace graph critical path

3) Reduce group

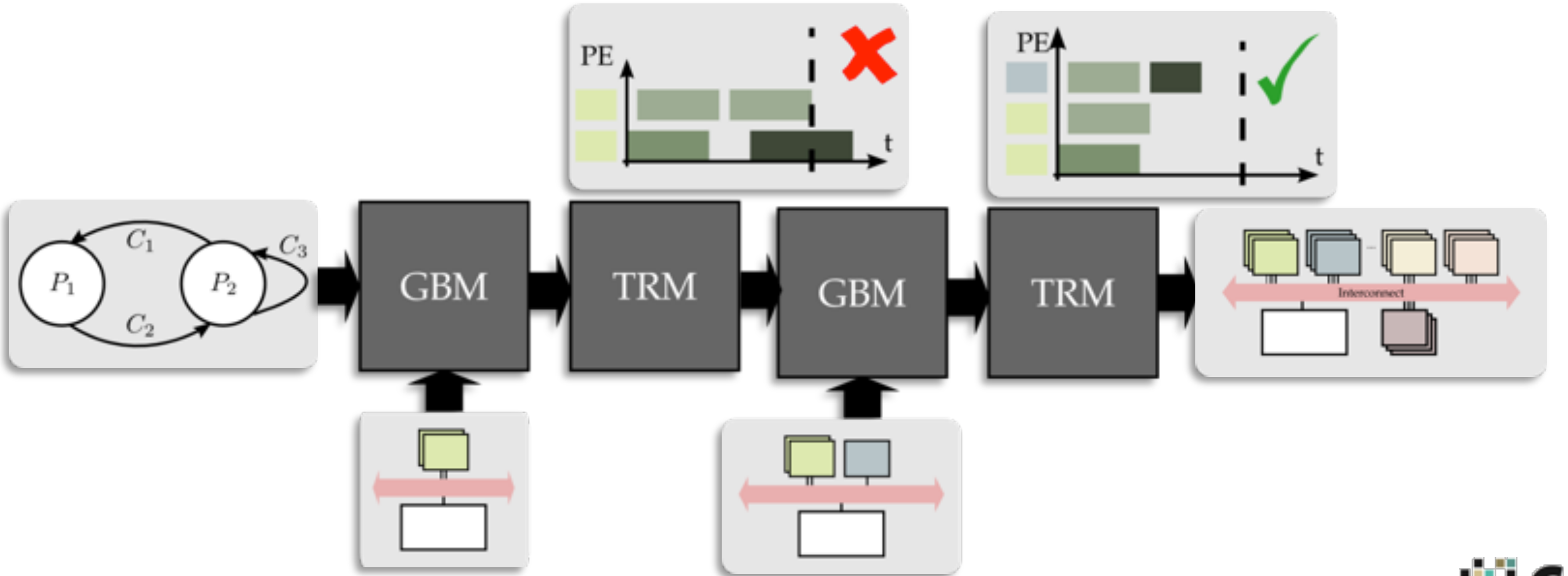
4) Assess & propagate

5) Quasi-homogeneous

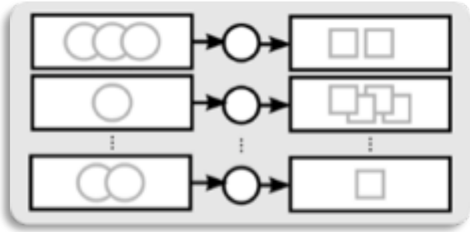


Heuristics for real-time applications

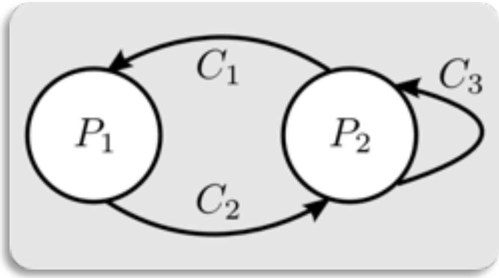
- Common approach: iteratively add resources to the mapping algorithm
  - Allocate more memory to communication channels
  - Add more processors (*intelligently*)



Code generation



Mapping configuration



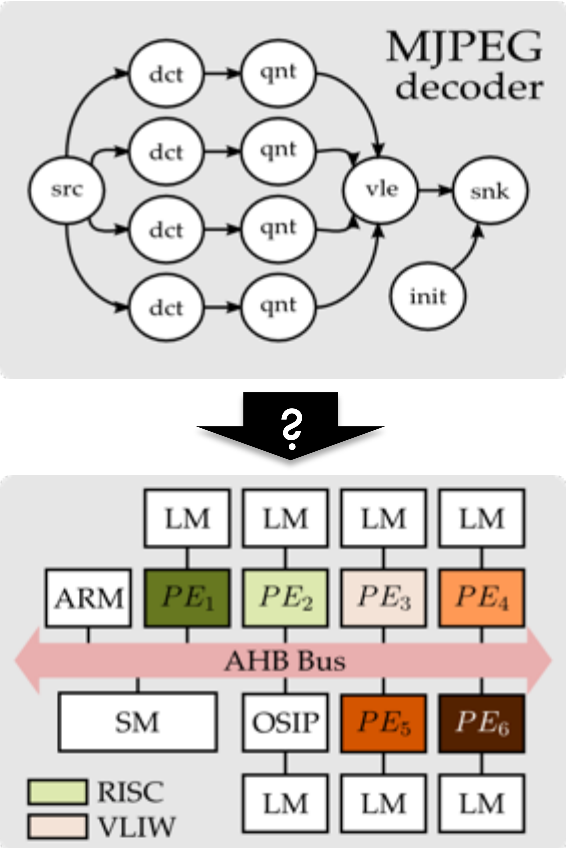
Application model

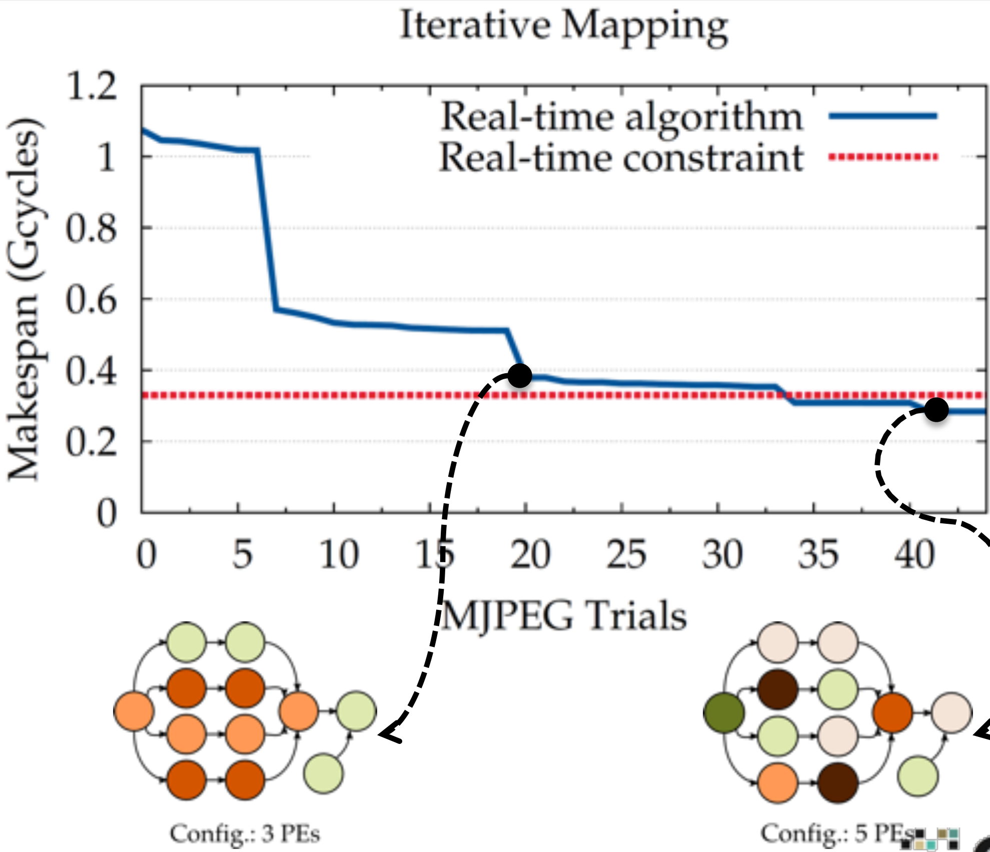
Code synthesis

Platform model (HW/SW)

```
PNargs_ifft_r.ID = 6U;
    PNargs_ifft_r.PNchannel_freq_coef = filtered_coef_right;
    PNargs_ifft_r.PNnum_freq_coef = 0U;
    PNargs_ifft_r.PNchannel_time_coef = sink_right;
    PNargs_ifft_r.channel = 1;
    sink_left = IPC1lmrf_open(3, 1, 1);
    sink_right = IPC1lmrf_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_FuncPtr)PNwra
pped_SourceTempl, &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_FuncPtr)PNwra
pped_radix2_1024_fft_Templ, &taskParams, &eb);
    glob_proc_cnt++;
    hasProcess = 1;
    taskParams.arg0 = (xdc_UArg)&PNargs_ifft_r;
    taskParams.priority = 1;

...
```

Sample results from mapping exploration





KPN Mapping: Lessons learned

- Explicit parallelism makes it easier
- Expressiveness requires intelligent trace analysis
- Performance estimation/measurement/simulation/prediction is important
- Code generation for heterogeneous MPSoCs bring great productivity improvements

- Research directions
  - HW acceleration in the input language and programming methodology
  - Mixing implicit parallelism to scale to many-cores
  - Adaptability: Modifications to the topology & mapping configuration
  - Energy-aware @ coarse-level

4. Summary

In this presentation

- Basics topics on compilers
- Efforts to hide complexity of programming heterogeneous multi and many-cores
  - With C code
  - From KPN extensions to C code
- Outlook
  - Raise abstraction further: More implicit parallelism
  - **But** give more information to compiler: Domain Specific Languages, for portable performance
  - Right balance: Compile vs. run-time
  - New goals: Energy efficiency & resilience

Thanks for the attention!

Questions?

References

- J. Ceng, J. Castrillón, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda, "MAPS: An integrated framework for MPSoC application parallelization," in Proceedings of the Design Automation Conference, 2008.
- J. Castrillón and R. Leupers, Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap. Springer, 2014.
- J. Castrillón, R. Leupers, and G. Ascheid, "MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs," IEEE Transactions on Industrial Informatics, vol. 9, no. 1, pp. 527–545, 2013.
- J. Castrillon, S. Schürmans, A. Stulova, W. Sheng, T. Kempf, R. Leupers, G. Ascheid, and H. Meyr, "Component-based waveform development: The nucleus tool flow for efficient and portable SDR," Wireless Innovation Conference and Product Exposition (SDR), 2010
- J. Castrillón, A. Tretter, R. Leupers, and G. Ascheid, "Communication-aware mapping of KPN appli- cations onto heterogeneous MPSoCs," in Proceedings of the Design Automation Conference, 2012.
- J. Castrillón, W. Sheng, and R. Leupers, "Trends in embedded software synthesis," in Proceedings of the International Conference on Embedded Computer Systems: Architecture, Modeling and Simulation, 2011.
- R. Leupers and J. Castrillón, "MPSoC programming using the MAPS compiler," in Proceedings of the Asia and South Pacific Design Automation Conference, 2010.
- R. Leupers, W. Sheng, and J. Castrillón, "Software compilation techniques for MPSoCs," in Handbook of Signal Processing Systems, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds. Springer, 2010, pp. 639–678.
- Weihua Sheng, Stefan Schürmans, Maximilian Odendahl, Mark Bertsch, Vitaliy Volevach, Rainer Leupers, and Gerd Ascheid. 2014. A compiler infrastructure for embedded heterogeneous MPSoCs. *Parallel Comput*. 40, 2 (February 2014), 51-68. DOI=10.1016/j.parco. 2013.11.007 http://dx.doi.org/10.1016/j.parco.2013.11.007