Optimizing Compilers

High Performance Computing, Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Outline

- Introduction to compilers
- Domain Specific Languages
- Intermediated Representations
- LLVM

What is a Compiler?

"a computer program that translates a program written in a high-level language into another language, usually machine language"

source: http://dictionary.reference.com

- German translations (from http://dict.leo.org)
  - Der Kompilierer
  - Der Übersetzer
    - Very interesting translation
    - Literally: translator

What is a Compiler?

Human readable notation → Program → Something else

- What is a compiler?

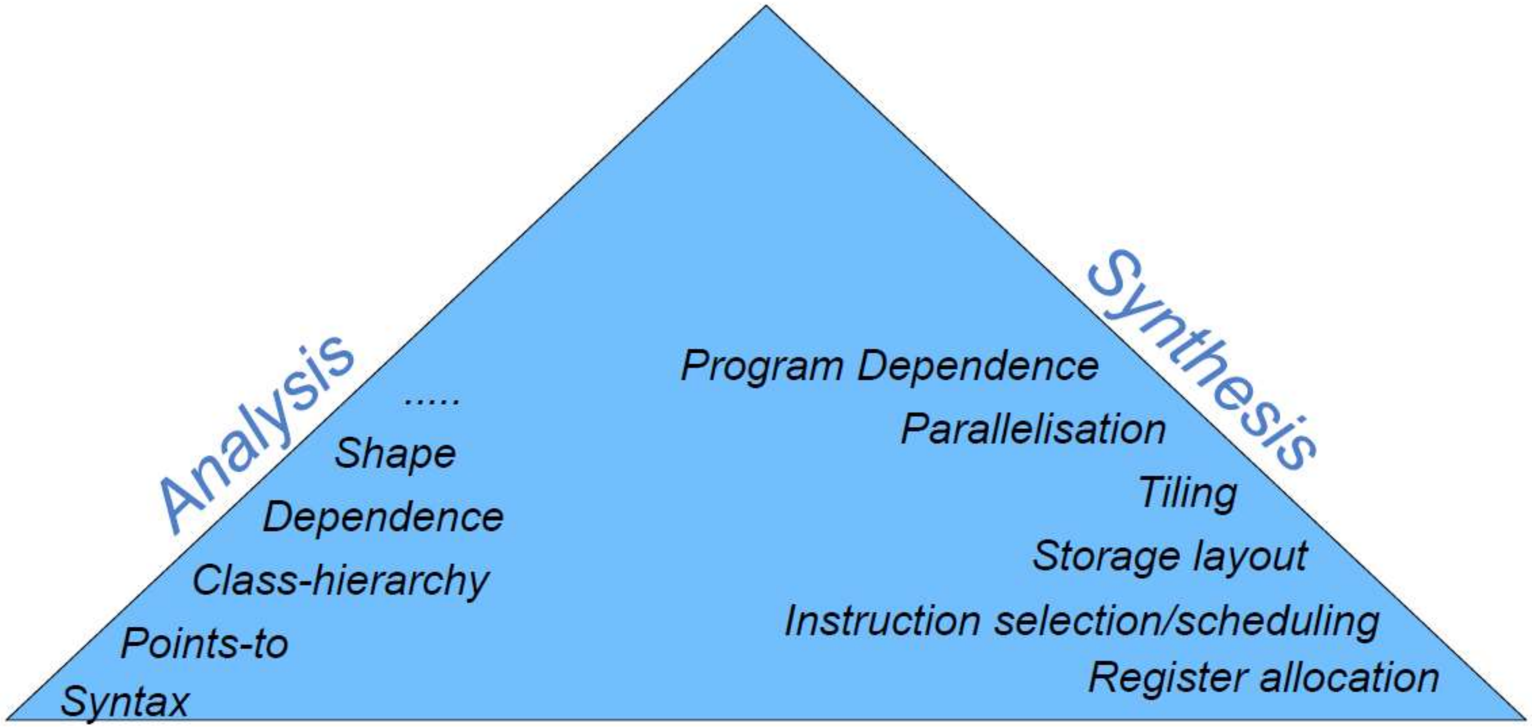
  a program that accepts as input a program text in a certain language and produces as output a program text in another language, while preserving the meaning of that text [Grune et al., 2000]

  a program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language) [AHO]

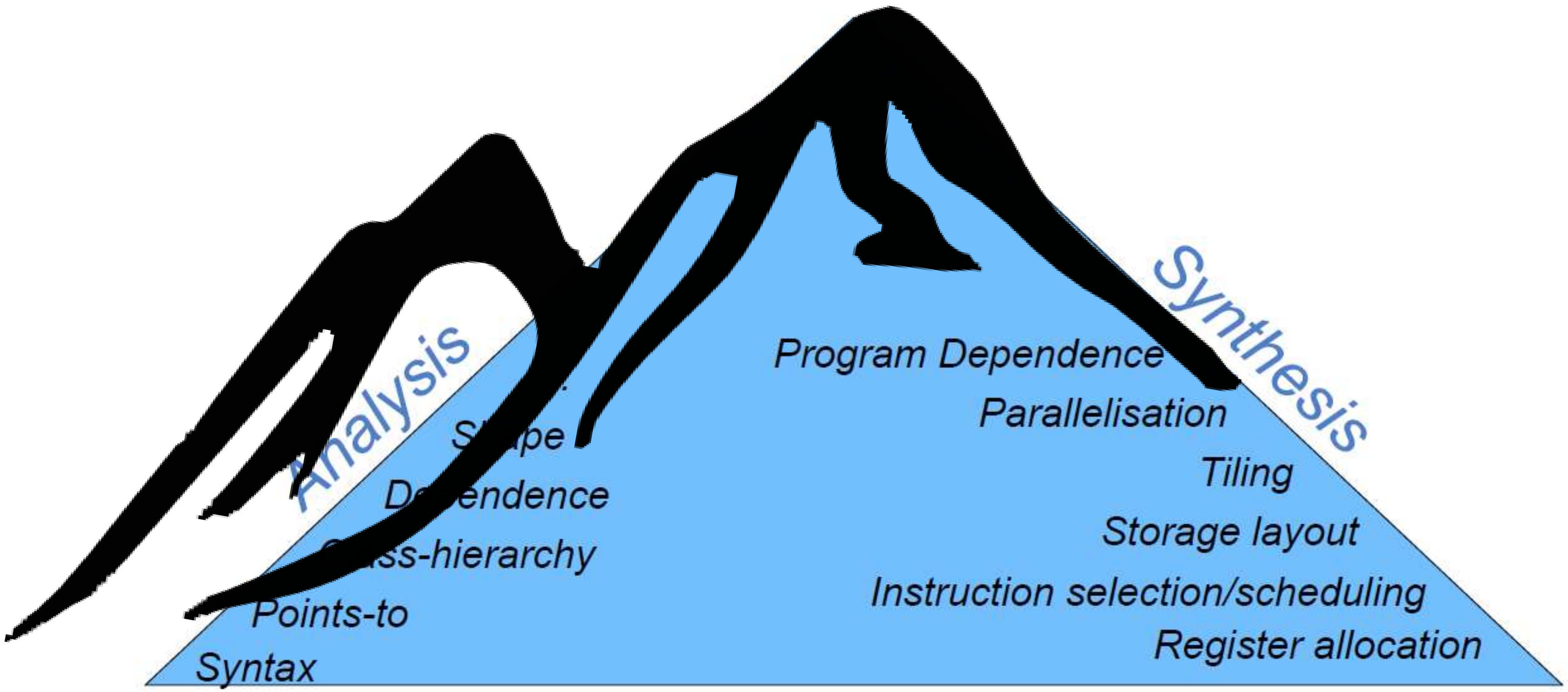
  - key: ability to extract properties of a source program (analysis) and transform it to construct a target program (synthesis)

- What is an interpreter?
  - a program that reads a source program and produces the results of executing this source
  - an interpreter directly executes, i.e. performs, instructions written in a programming language, without previously compiling them into a machine language program

Analysis & Synthesis



Analysis & Synthesis



Courtesy of Paul Kelly, Imperial College London

Example 1: Source-to-source Compilers

- Also called transpiler or transcompiler
  - e.g., C-to-C
- Code transformation at source level
  - e.g., automatic parallelization, data layout transformations, …
- High-level intermediate representation
- Examples
  - Rose, Insieme, Pluto, Cetus, …

Example 2: Java JIT Compiler

- Java compiler
  - the output is a class file (.class)
  - `javac` (Oracle), `gcj` (GNU Compiler for Java), ECJ (Eclipse for Java), …
  - platform-neutral Java bytecode
  - there are also compilers that emit optimized native machine code for a particular hardware/operating system combination
- Most Java-to-bytecode compilers do little optimization, leaving this to the JRE (Java Runtime) at runtime
- Just-in-time (JIT) compilation
  - the Java virtual machine (JVM) loads the class files and either interprets the bytecode or just-in-time compiles it to machine code and then possibly optimizes it using dynamic compilation.
  - interaction between JVM and Java compilers specified in JSR 199

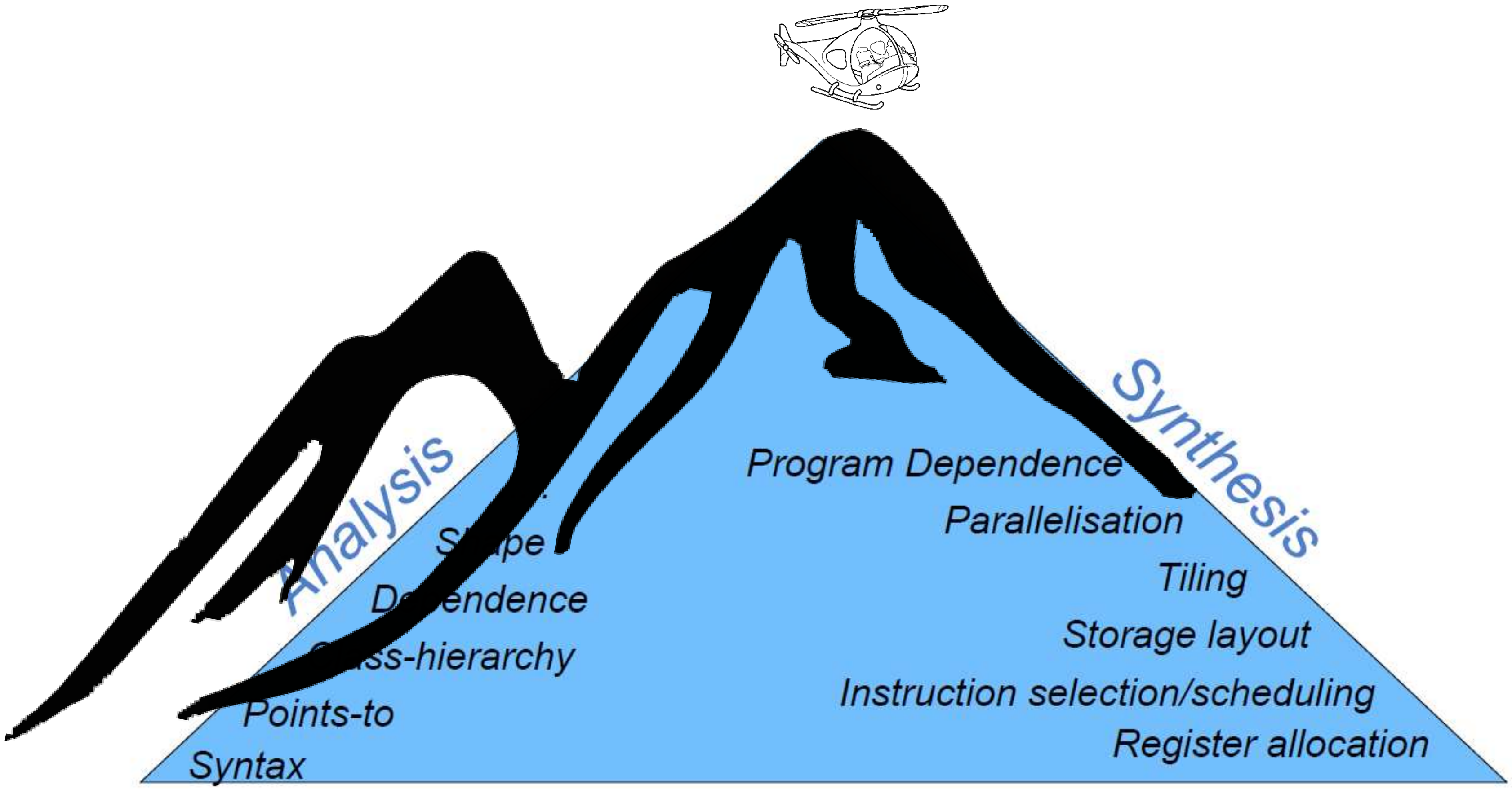
Example 3: Domain-Specific Languages

- Input is a domain-specific language (DSL)
  - a language with domain-specific construct and constraints
- Assumptions (and restriction) on the input
  - analysis simpler
  - optimization is relatively easier
    - Domain-specific optimization
- Examples
  - OpenGL Shading Language
  - Halide for image processing
  - SQL for database

Example: simple GLSL fragment shader

```
varying vec3 N;
varying vec3 v;

void main(void){
  vec3 L = normalize(gl_LightSource[0].position.xyz-v);
  vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0);
  Idiff = clamp(Idiff, 0.0, 1.0);
  gl_FragColor = Idiff;
}
```

DSL: - Analysis, + Synthesis



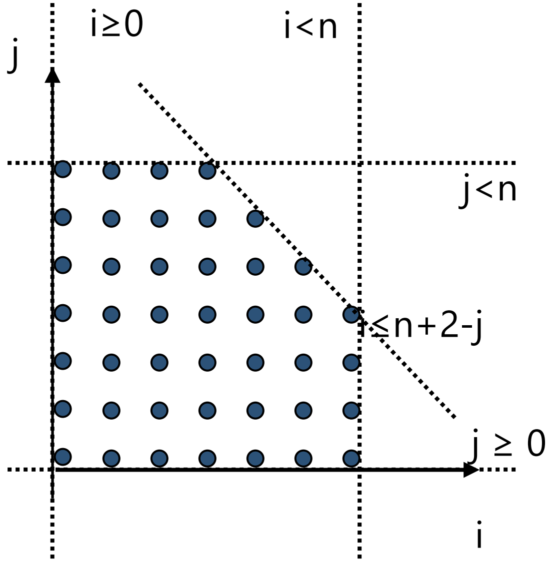
Courtesy of Paul Kelly, Imperial College London

Example 4: Parallelizing Compilers

- Input is sequential code
- Output is parallel code, i.e., expose some kind of parallelism
- Typically, parallelism is extracted from loops
  - advanced analysis, e.g., using the polyhedral model
  - `#pragma` notations to help the compiler job
- Sometime parallelizing compilers enhance parallelization
  - E.g., from shared memory parallel code to distributed or heterogeneous systems
- Form of parallelism
  - automatic vectorization (SIMD instructions), e.g., by `gcc`, `llvm` and `icc`
  - multi-threading (pthread), e.g., by Rose, Pluto, Insieme, LLVM-Polly
  - distributed memory (typically MPI)

Automatic Parallelization with the Polyhedral Model

```
for(int i=0; i<n; i++)
  for(int j=0; j<n; j++)
    if(i <= n+2-j)
(s)     b[j] = b[j] + a[i];
```



Polyhedron for n=6

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ n-1 \\ 0 \\ n-1 \\ -n-2 \end{pmatrix} \geq \vec{0}$$

Iteration domain of S

$$\begin{array}{cccc} i & j & n & \text{constant} \\ \Downarrow & \Downarrow & \Downarrow & \Downarrow \end{array}$$

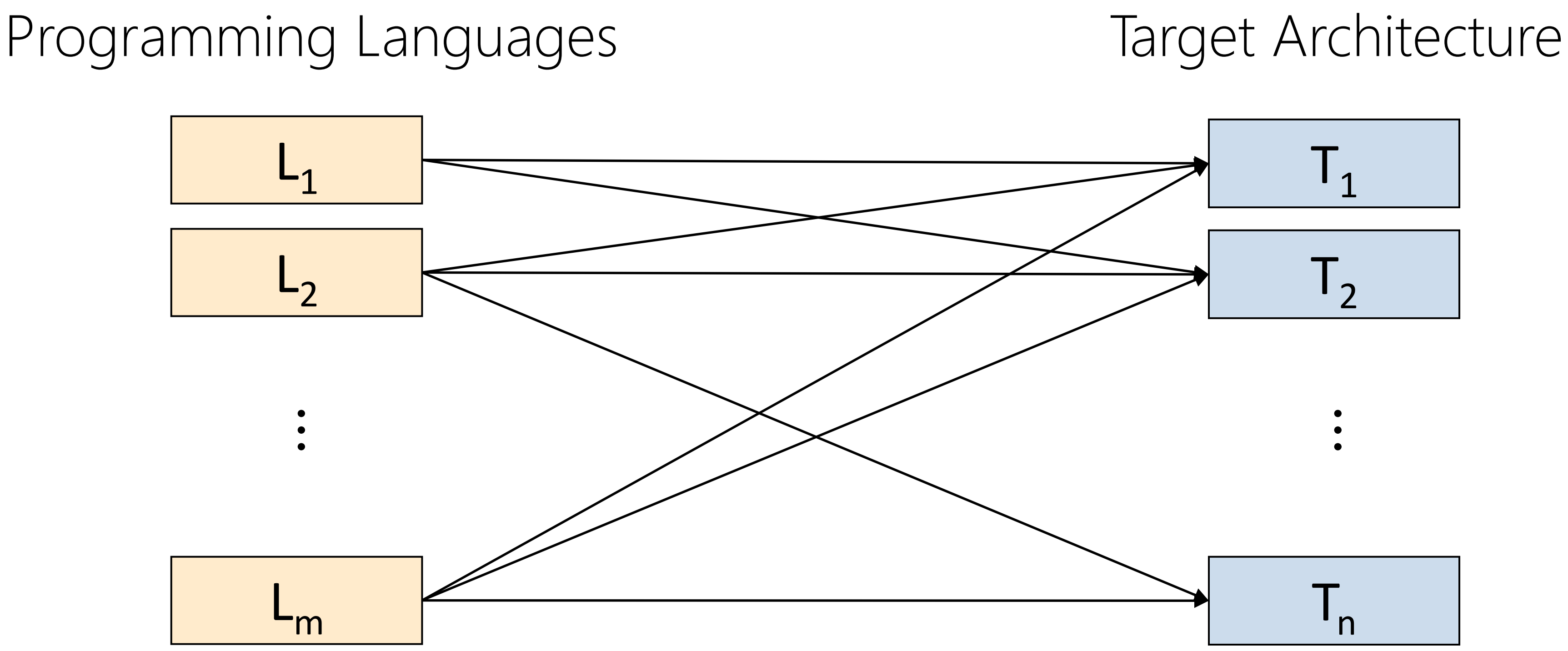
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & -1 \\ -1 & -1 & -1 & -2 \end{bmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \geq \vec{0}$$

Iteration domain with homogenous coordinates

From Languages to Target Architectures

Programming Languages	Target Architecture
L_1	T_1
L_2	T_2
⋮	⋮
L_m	T_n

From Languages to Target Architectures



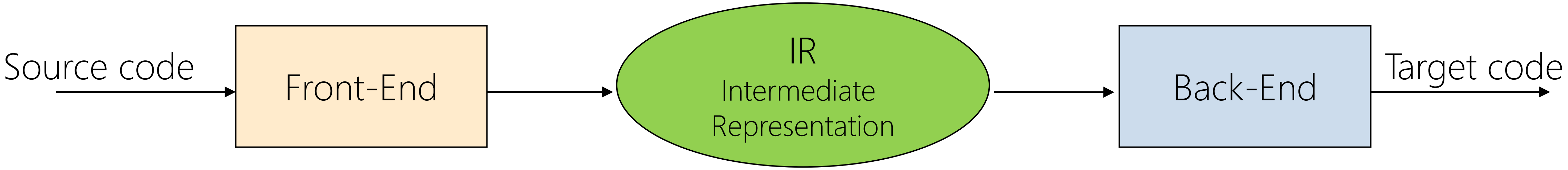
General Structure of a Compiler

Front-end performs the analysis of the source language

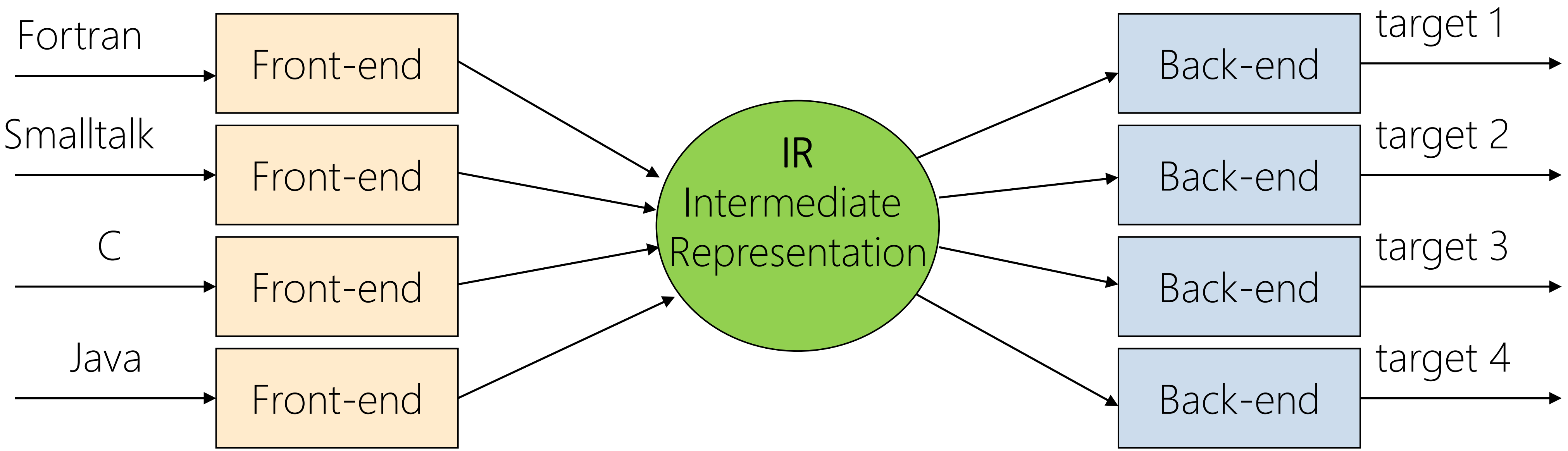
- recognizes legal and illegal programs and reports errors
- understands the input program and collects its semantics in an Intermediate Representation (IR)
- produces IR and shapes the code for the back-end

Back-end does the target language synthesis

- chooses instructions to implement each IR operation
- translates IR into target code
- needs to conform with system interfaces
- automation has been less successful

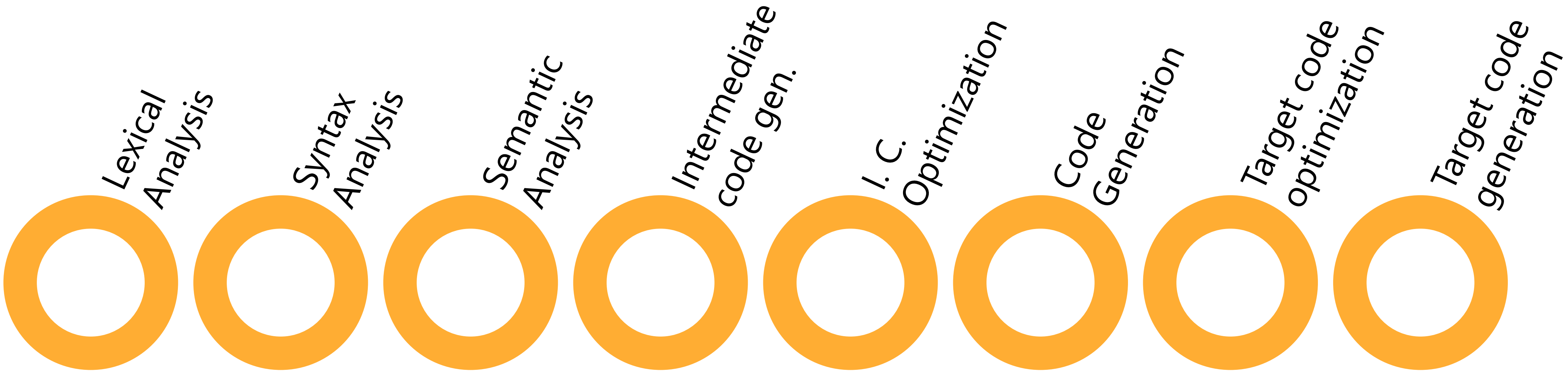


mxn compilers with m+n components

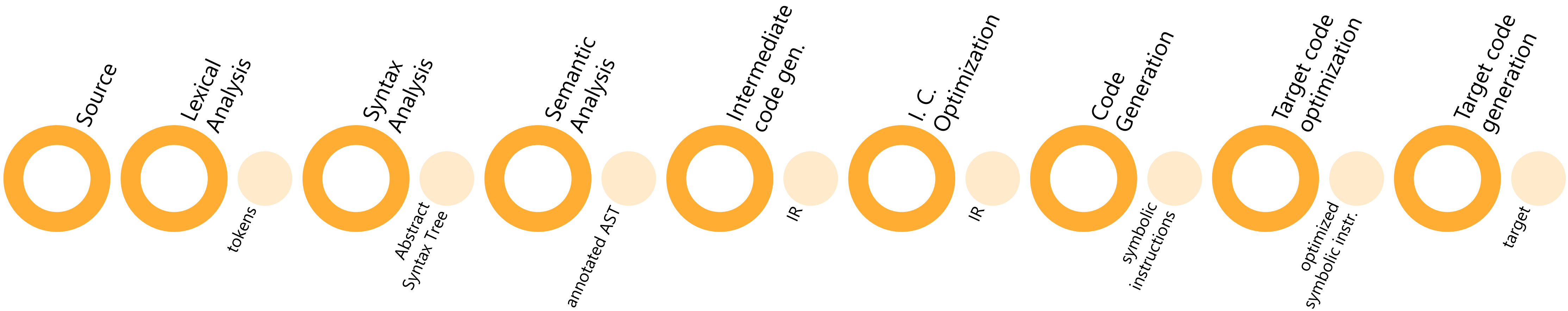


- All language-specific knowledge must be encoded in the front-end
- All target-specific knowledge must be encoded in the back-end

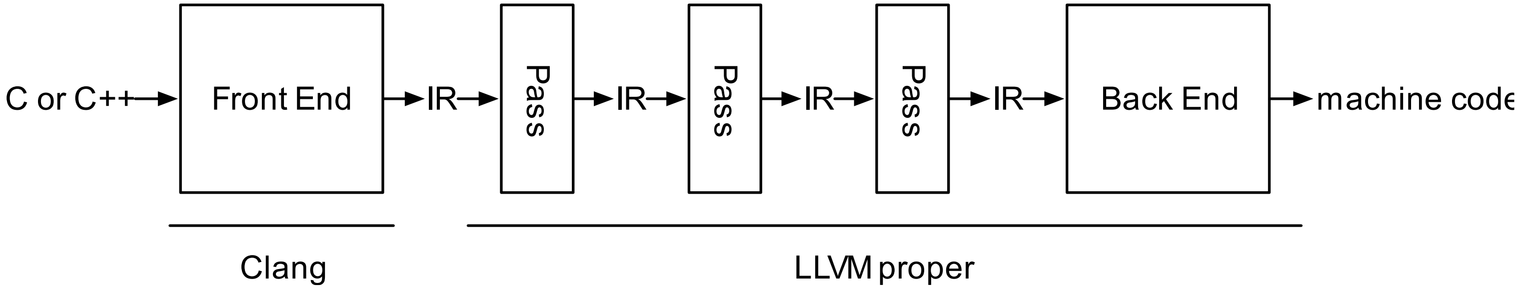
General Compiler Structure



General Compiler Structure



- Example: the LLVM compiler



Lexical Analysis

- Reads characters in the source program and groups them into words (basic unit of syntax)
- Produces words and recognizes what sort they are
- The output is called token and is a pair of the form `<type,lexeme>` or `<token_class, attribute>`
  - E.g.: **`a=b+c`** becomes `<id,`**`a`**`> <=,> <id,`**`b`**`> <+,> <id,`**`c`**`>`
- Needs to record each id attribute: keep a symbol table
  - Lexical analysis eliminates white space, etc
- Speed is important - use a specialized tool
  - e.g., Flex: a tool for generating scanners: programs which recognize lexical patterns in text

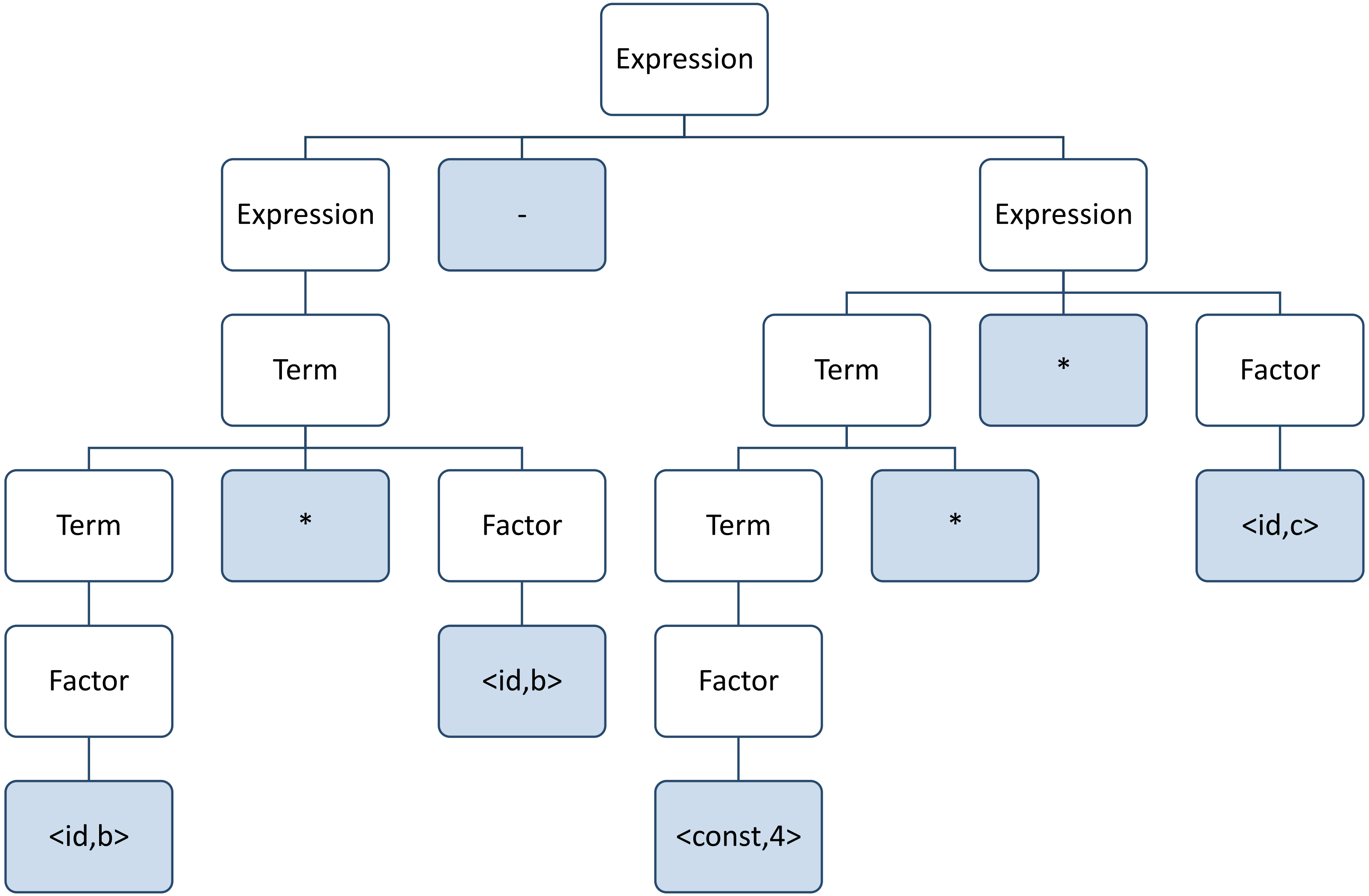
Syntax Analysis (Parsing)

- Imposes a hierarchical structure on the token stream
- This hierarchical structure is usually expressed by recursive rules
- Context-free grammars formalise these recursive rules and guide syntax analysis
- Example
  - A grammar defining simple algebraic expressions

```
expression  -> expression '+' term | expression '-' term | term
term -> term '*' factor | term '/' factor | factor
factor -> identifier | constant | '(' expression ')'
```

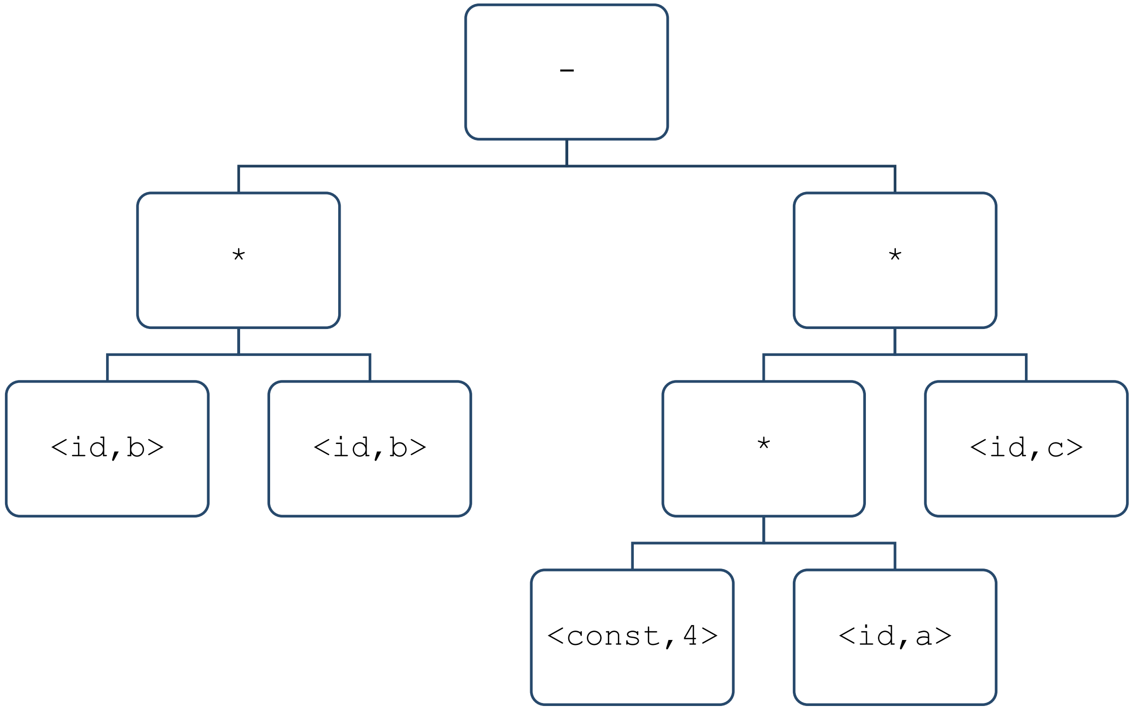
Parsing

- Parse tree for
  `b*b-4*a*c`



Abstract Syntax Tree (AST)

- Example: an AST for `b*b-4*a*c`
- An Abstract Syntax Tree (AST) is a more useful data structure for internal representation
  - a compressed version of the parse tree
    - summary of grammatical structure without details about its derivation
  - ASTs are a form of IR
- Used as fundamental representation in source-to-source compiler



AST Example: Clang AST

- Example code:

```
int f(int x) {
  int result = (x / 42);
  return result;
}
```

- Compile with:

```
clang -Xclang -ast-dump -fsyntax-only test.cc
```

- Output:

```
TranslationUnitDecl 0x5aea0d0 <<invalid sloc>>
... cutting out internal declarations of clang ...
`-FunctionDecl 0x5aeab50 <test.cc:1:1, line:4:1> f 'int (int)'
  |-ParmVarDecl 0x5aeaa90 <line:1:7, col:11> x 'int'
  `-CompoundStmt 0x5aead88 <col:14, line:4:1>
    |-DeclStmt 0x5aead10 <line:2:3, col:24>
    | `-VarDecl 0x5aeac10 <col:3, col:23> result 'int'
    |   `-ParenExpr 0x5aeacf0 <col:16, col:23> 'int'
    |     `-BinaryOperator 0x5aeacc8 <col:17, col:21> 'int' '/'
    |       |-ImplicitCastExpr 0x5aeacb0 <col:17> 'int' <LValueToRValue>
    |       | `-DeclRefExpr 0x5aeac68 <col:17> 'int' lvalue ParmVar 0x5aeaa90 'x' 'int'
    |       `-IntegerLiteral 0x5aeac90 <col:21> 'int' 42
    `-ReturnStmt 0x5aead68 <line:3:3, col:10>
      `-ImplicitCastExpr 0x5aead50 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0x5aead28 <col:10> 'int' lvalue Var 0x5aeac10 'result' 'int'
```

AST Example: dHPF High Performance Fortran

- Fortran code with parallel annotations

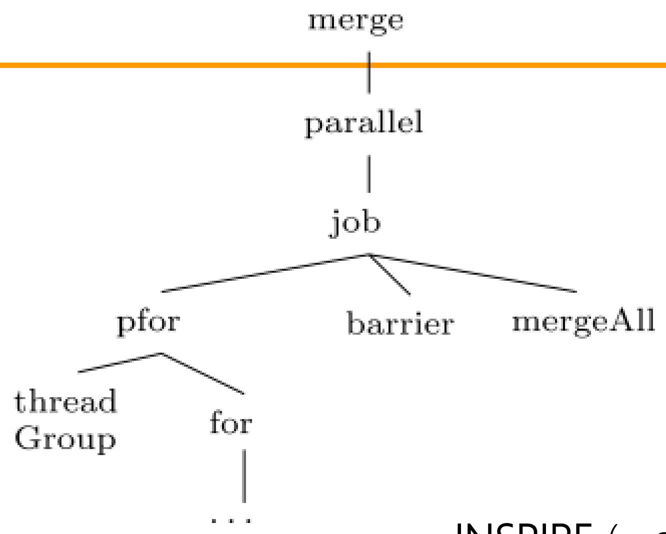
```
PROGRAM MAIN
      REAL A(100), X
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
      FORALL (i=1:100) A(i) = X+1
                    CALL FOO(A)
      END

      SUBROUTINE FOO(X)
      REAL X(100)
!HPF$ INHERIT X
      IF (X(1).EQ.0) THEN
        X = 1
      ELSE
        X = X + 1
      END IF
      RETURN
      END
```

```
(1[GLOBAL]                              ! 2 components in true-branch
   ((2[PROG_HEDR]                       ! 6 components in true-branch
      (3[VAR_DECL]
       4[PROCESSORS_STMT]
       5[DISTRIBUTE_DECL]
        (6[FORALL_STMT]
           (7[ASSIGN_STAT]
            8[CONTROL_END]
           )
           NULL
        )
       9[PROC_STAT]
       10[CONTROL_END]
      ) NULL
    )
    (11[PROC_HEDR]
       (12[VAR_DECL]
        13[INHERIT_DECL]
        (14[LOGIF_NODE]                  ! both branches are non empty
           (15[ASSIGN_NODE]
            16[CONTROL_END]
           )
           (17[ASSIGN_NODE]
            18[CONTROL_END]
           )
        )
        19[RETURN_STAT]
        20[CONTROL_END]
       ) NULL
    ) NULL
)
```

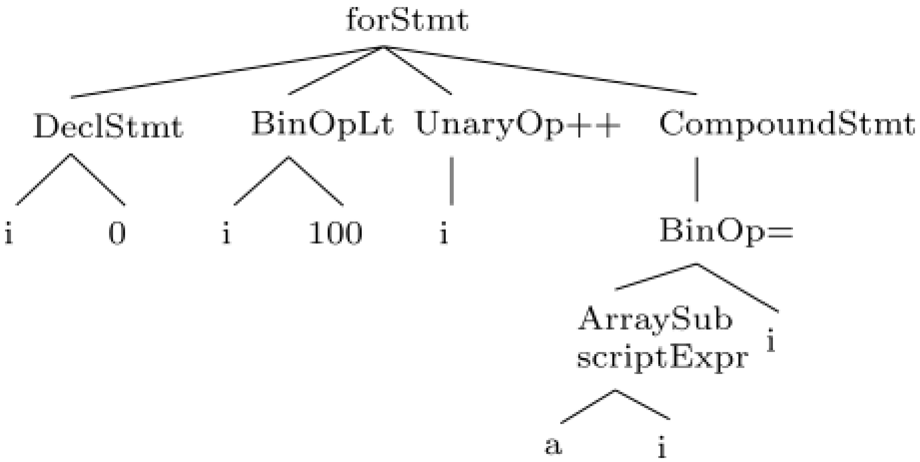
AST Example: the Insieme Parallel Compiler

- Source-to-source compiler, parallel by design
- INSPIRE (INSieme Parallel Intermediate Representation)
  - high-level IR
  - all transformations happens at AST level
  - output is C/C++ code enabling parallelization with pthreads, MPI, OpenCL
  - parallelism is expressed in the IR
- Example

```
int a [N];
#pragma omp parallel
{
  #pragma omp for
  for(int i=0; i<N; i++){
    a[i]=i;
  }
}
```



INSPIRE (--omp-sema)



Clang AST(-Xclang -ast-dump)

Semantic Analysis (Context Handling)

- Collects context (semantic) information, checks for semantic errors, and annotates nodes of the tree with the results
- Examples
  - type checking: report error if an operator is applied to an incompatible operand
  - check flow-of-controls
  - uniqueness or name-related checks

Intermediate Code Generation

- Translate language-specific constructs in the AST into more general constructs
- Example of a form of IR (3-address code)

```
tmp1 = 4
tmp2 = tmp1*a
tmp3 = tmp2*c
tmp4 = b*b
tmp5 = tmp4-tmp3
```

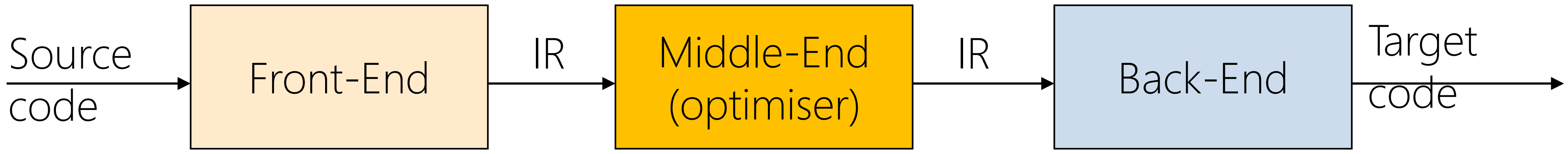
- Typically organized in a control flow graph

Control Flow Graph

- Control-flow graph (CFG): a graph-based representation of all paths that might be traversed through a program during its execution
  - each node is an instruction or a basic block
  - edges represent jump in the control flow
- Basic block
  - straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - only one entry block, only one exit block

Code Optimization

- The goal is to improve the intermediate code and, thus, the effectiveness of code generation and the performance of the target code
- Optimizations can range from trivial (e.g., constant folding) to highly sophisticated (e.g., in-lining)
  - Example: replace the first two statements in the example of the previous slide with: `tmp2=4*a`
- Modern compilers perform such a range of optimizations, that one could argue for:



Optimizations

- Example: Dead Code Elimination (DCE)

```
int global;
void f ()
{
  int i;
  i = 1;
  global = 1;
  global = 2;
  return;
  global = 3;
}
```

```
int global;
void f ()
{



  global = 2;
  return;

}
```

Code Generation

- Map the IR onto a linear list of target machine instructions in a symbolic form
- Three major steps
  1. instruction selection: a pattern matching problem
  2. register allocation: each value should be in a register when it is used (but there is only a limited number): NP-Complete problem
  3. instruction scheduling: take advantage of multiple functional units: NP-Complete problem
- Target, machine-specific properties may be used to optimize the code
- Finally, machine code and associated information required by the Operating System are generated

Other Representations used in Compilers

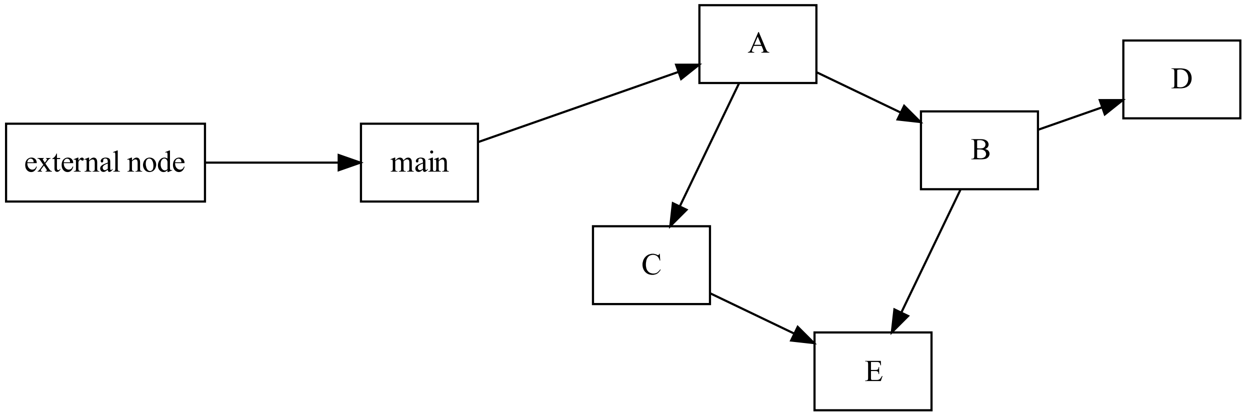
- Call graph
  - shows dependences between procedures
  - useful for inter-procedural analysis and optimization
  - E.g., inlining using Strongly Connected Component (Tarjan's algorithm)
- Data dependence graph
  - encodes the flow of data
  - node: program statement
  - edge: connects two nodes if one uses the result of the other
  - useful in examining the legality of program transformations

Example: Call Graph in Clang/LLVM

- Example code:

```
static void E() { }
static void D() { }
static void C() { E(); }
static void B() { D(); E(); }
static void A() { B(); C(); }
int main() {
  A();
}
```

- Output (circo svg engine):



Call graph

- Compile: `clang -S -emit-llvm main.c -o - | opt -analyze -dot-callgraph`
- Visualize: `dot -Tpng -ocallgraph.png callgraph.dot`

Getting Started with LLVM

- You can generate LLVM IR (also called bitcode) with the following command

```
clang example.c -o example.bc -c -emit-llvm
```

- you can quickly run it by using the LLVM interpreter

```
lli example.bc
```

- The bitcode can be browsed by using the LLVM disassembler

```
llvm-dis < example.bc
```

- To produce assemply from the bytecode with the LLVM backend compiler

```
llc example.bc -o example.s
```

LLVM Basic Block Example

- Code example

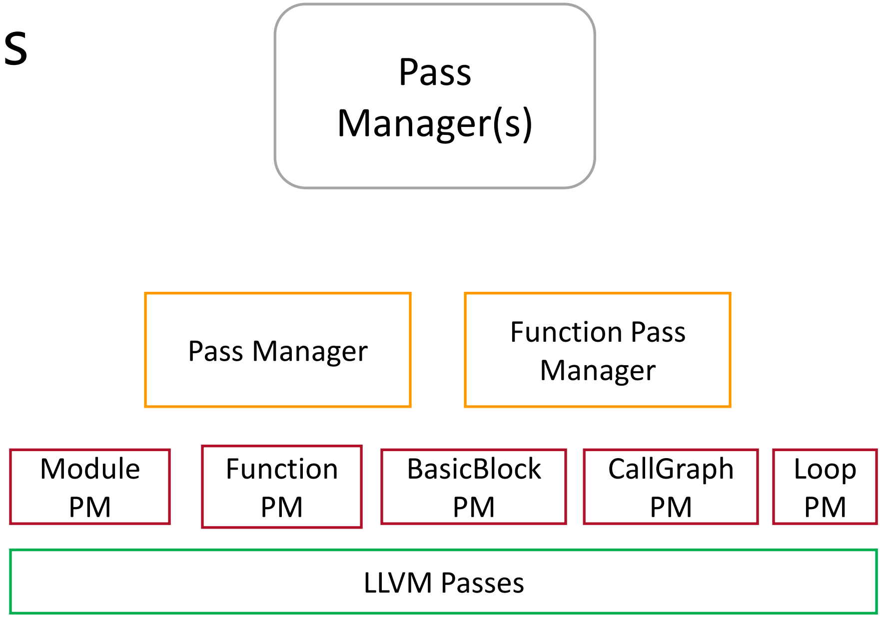
```
void test(bool b) {
    int a;
    if (b) {
        a = 1;
    }
    printf("%d\n",a);
}
```

- LLVM bitcode

```
define void @test(i1 zeroext) #0 !dbg !6 {
  %2 = alloca i8, align 1
  %3 = alloca i32, align 4
  %4 = zext i1 %0 to i8
  store i8 %4, i8* %2, align 1
  call void @llvm.dbg.declare(metadata i8* %2, metadata !10, metadata !11), !dbg !12
  call void @llvm.dbg.declare(metadata i32* %3, metadata !13, metadata !11), !dbg !15
  %5 = load i8, i8* %2, align 1, !dbg !16
  %6 = trunc i8 %5 to i1, !dbg !16
  br i1 %6, label %7, label %8, !dbg !18

; <label>:7:                                      ; preds = %1
  store i32 1, i32* %3, align 4, !dbg !19
  br label %8, !dbg !21

; <label>:8:                                      ; preds = %7, %1
  %9 = load i32, i32* %3, align 4, !dbg !22
  %10 = call i32 (i8*, ...) @printf(...), !dbg !23
  ret void, !dbg !24
}
```

The LLVM Compilation Infrastructure

- We can build compiler passes at different levels
  - Basic block
  - Function
  - Loop
  - Region
  - Call graph



LLVM BasicBlock Pass

- An example BB pass that prints all instructions

```
#include <llvm/IR/Type.h>
#include <llvm/Pass.h>
#include <llvm/IR/BasicBlock.h>
#include <llvm/Support/raw_ostream.h>
using namespace llvm;
namespace {
class MyBlockPass : public BasicBlockPass {
public:
    static char ID;
    MyBlockPass() : BasicBlockPass(ID) {}

    virtual bool runOnBasicBlock(BasicBlock &BB) {
      errs().write_escaped(BB.getName()) << '\n';
      // iterating instructions in the current BasickBlock
      for(Instruction &i : BB){
          errs() << " - "<< i.getOpcodeName() << " ";
          Type *type = i.getType();
          type->print(errs());
          errs() << '\n';
      }
      errs() << '\n';
      return false;
    }
  };
} // namespace

char MyBlockPass ::ID = 0;
static RegisterPass<MyBlockPass > X("block-pass", "Block Pass");
```

LLVM Function Pass

```
#include <llvm/Pass.h>
#include <llvm/IR/Function.h>
#include <llvm/Support/raw_ostream.h>
using namespace llvm;
namespace {
class MyFunctionPass  : public FunctionPass {
public:
    static char ID;
    MyFunctionPass  () : FunctionPass(ID) {}
    virtual bool runOnFunction(Function &F) {
        errs().write_escaped(F.getName()) << '\n';
        // iterate arguments
        errs() << " - args:" << '\n';
        for(Argument &a : F.getArgumentList()){
            errs() << "   - ";
            errs().write_escaped(a.getName()) << '\n';
        }
        // iterate BB in a function
        errs() << " - blocks:" << '\n';
        for(BasicBlock &b : F.getBasicBlockList()){
            errs() << "   * ";
            errs().write_escaped(b.getName()) << '\n';
        }
        errs() << '\n';
        return false;
    }
};
} // namespace
char MyFunctionPass::ID = 0;
static RegisterPass<MyFunctionPass> X("function-pass", "FunctionPas
```

LLVM Loop Pass

- Warning: there is not explicit loop in a CGF
- Only works after Loop Analysis pass
  - Identifies natural loop

```
#include <llvm/Analysis/LoopPass.h>
#include <llvm/Support/raw_ostream.h>
using namespace llvm;

namespace {

class MyLoopPass : public LoopPass {
public:
    static char ID;
    MyLoopPass() : LoopPass(ID) {}

    bool runOnLoop(Loop *L, LPPassManager &LPPM) {
      L->print(errs());
      for(BasicBlock *b : L->getBlocks()) {
          errs() << " - ";
          errs().write_escaped(b->getName()) << '\n';
      }
      errs() << '\n';
      return false;
    }
};
}// namespace

char MyLoopPass::ID = 0;
static RegisterPass<MyLoopPass> X("loop-pass", "Loop Pass");
```

What Compilation Passes are Executed with Clang/LLVM?

- Basic passes (no opt)

```
clang -mllvm -opt-bisect-limit=-1 test.cc
```

- O3 passes

```
clang -O3 -mllvm -opt-bisect-limit=-1 test.cc
```

Lab

- Experiment with LLVM
  - Generate AST and LLVM bitcode code for C/C++ code
  - Compare LLVM bitcode with different compilation flags

Instruction Scheduling, Compiler-based Optmizations

High Performance Computing, Summer 2021

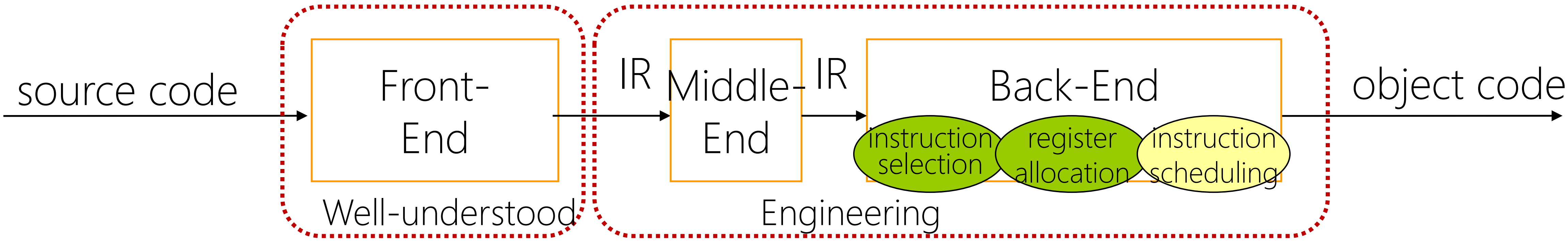
Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Outline

- Instruction scheduling
- Optimization overview
  - Loop optimizations

Instruction Scheduling

- The problem
  - given a code fragment for some target machine and the latencies for each individual operation, reorder operations to minimize execution time
  - recall: modern processors may have multiple functional units
- The task
  - produce correct code; minimize wasted cycles; avoid spilling registers; operate efficiently



Instruction Scheduling: Background

- Many operations have delay latencies for execution

E.g., load, store: <delay> CPU cycles (depends on the processor)

  - issue load, result appears <delay> cycles later
  - execution continues unless result is referenced
  - premature reference causes hardware to stall

- Modern machines can issue several operations per cycle (pipelining)
- Execution time is order-dependent (has been since the 60s)
- Overview of a solution
  - move loads back at least <delay> slots from where they are needed, but this increases register pressure (i.e., more registers may be needed)
  - ideally, we want to minimize both hardware stalls and added register pressure

Motivating Example

- Two variants to compute `(a+b)+c`
  - assume that the latency of a load is 3 cycles; all other instructions have a latency of 1 cycle
    - just an example with simplified hardware model
  - note: costs due to the memory hierarchy are not taken in account

9 cycles

```
load r1, @a
load r2, @b
add r1,r1,r2
load r3, @c
add r1,r1,r3
```

Pipeline stalls

6 cycles

```
load r1, @a
load r2, @b
load r3, @c
add r1,r1,r2
add r1,r1,r3
```

Instruction Scheduling Algorithm for a Basic Block

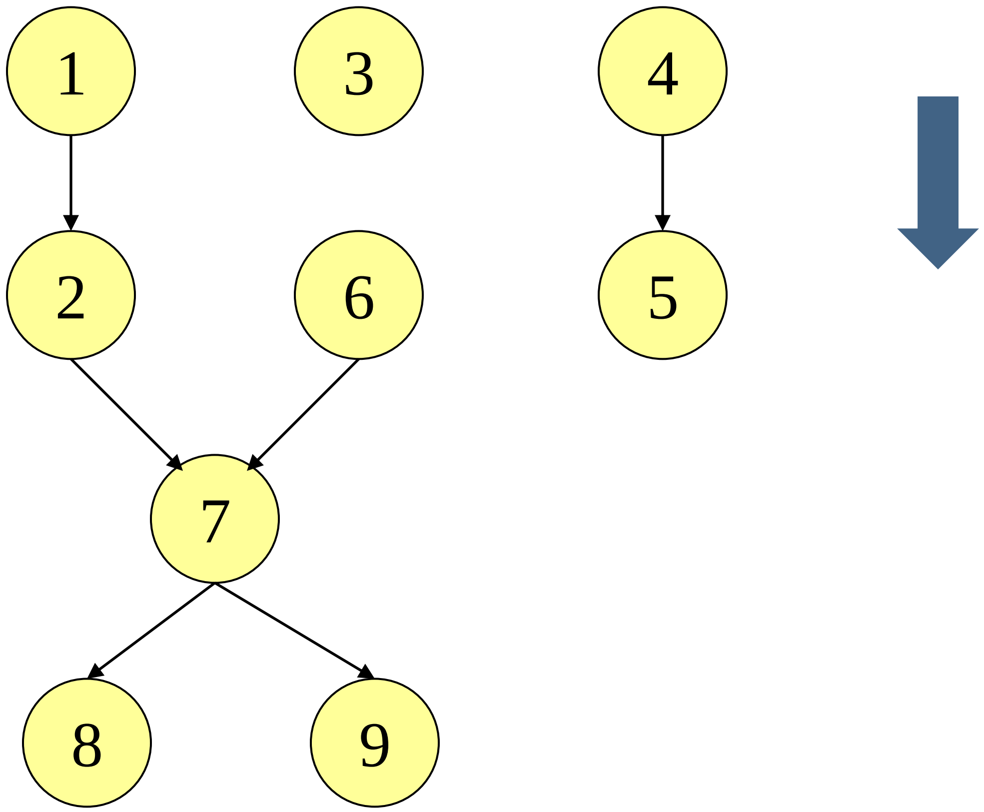
1. Build a precedence (data dependence) graph
2. Compute a priority function for the nodes of the graph
3. Use list scheduling to construct a schedule, one cycle at a time
   1. use a queue of operations that are ready
   2. at each cycle
      - choose a ready operation and schedule it
      - update the ready queue

- A greedy heuristic; open to variations
  - recall: greedy heuristic: an algorithmic technique in which an optimization problem is solved by finding locally optimal solutions

1. Build a Precedence Graph

- Dependence graph for the following code

```
1. load r1, @x
2. load r2, [r1+4]
3. and r3, r3, 0x00FF
4. mult r6, r6, 100
5. store r6
6. div r5, r5, 100
7. add r4, r2, r5
8. mult r5, r2, r4
9. store r4
```



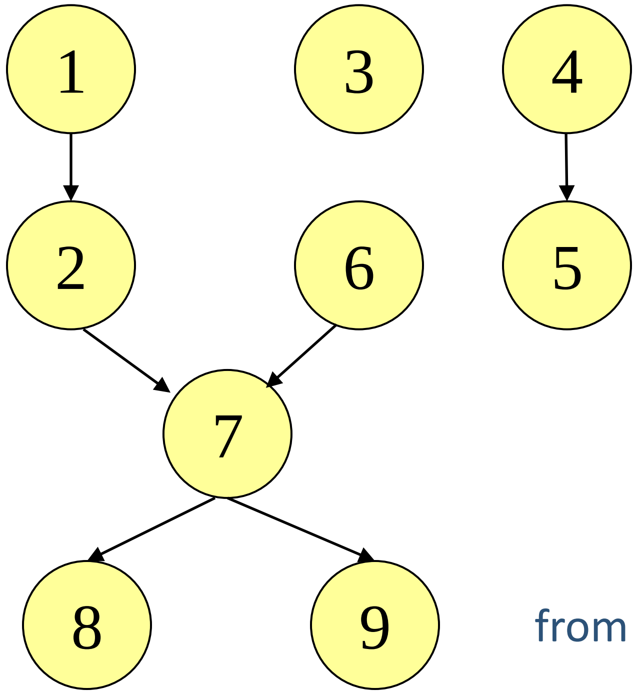
2. Compute a Priority Function

- Assign to each node a weight equal to the longest delay latency (total) to reach a leaf in the graph from this node (include latency of current node)

$$\text{weight}_i = \text{latency}_i + \max(\text{weight}_{\text{all successor nodes}})$$

- Assume
  - div 4 cycles
  - mult 3 cycles
  - 1 cycle for the rest

```
1.   load r1, @x
2.   load r2, [r1+4]
3.   and r3, r3, 0x00FF
4.   mult r6, r6, 100
5.   store r6
6.   div r5, r5, 100
7.   add r4, r2, r5
8.   mult r5, r2, r4
9.   store r4
```



bottom-up
from leaves to root

node	weight
1	6
2	5
3	1
4	4
5	1
6	8
7	4
8	3
9	1

3. Local List Scheduling Algorithm

```
Cycle=1; Ready = set of available operations; Active = {}
while (Ready U Active ≠ {})
      if (Ready ≠ {}) then
            remove an op from Ready (based on the weight)
            schedule(op) = cycle
            Active = Active U op
      cycle = cycle+1
      for each op in Active
            if (schedule(op) + delay(op) <= cycle) then
                  remove op from Active
                  for each immediate successor s of op
                        if (all operand of s are available) then
                              Ready = Ready U s
```

Removal in priority order

if op has completed execution

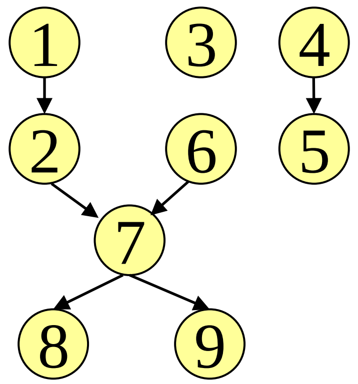
if all successor's op are ready,
then put it on ready

Finding the Schedule

- **Final schedule**

Cycle	Instructions ready	Schedule	Instructions active
1	6, 1, 4, 3	`div r5,r5,100`	6
2	1, 4, 3	`load r1,@x`	6, 1
3	2, 4, 3	`load r2,[r1+4]`	6, 2
4	4, 3	`mult r6,r6,100`	6, 4
5	7, 3	`add r4,r2,r5`	4, 7
6	8, 3, 9	`mult r5,r2,r4`	4, 8
7	3, 9, 5	`and r3,r3,0x00ff`	8, 3
8	9, 5	`store r4`	8, 9
9	5	`store r6`	5

node	weight
1	6
2	5
3	1
4	4
5	1
6	8
7	4
8	3
9	1



More on Scheduling

- Two distinct classes of list scheduling
  - forward list scheduling: start with all available operations; work forward in time (Ready: all operands available)
  - backward list scheduling: start with leaves; work backward in time (Ready: latency covers uses)
  - folk wisdom is to try both and keep the best result
- Variations on computing priority function
  - maximum path length containing node (decreases register usage)
  - prioritize critical path
  - number of immediate successors or total number of descendants
  - increment weight if node contains a last use (shortens live ranges)
  - do not add latency to node’s weight
  - maximum delay latency from first available node

Example: Extend List Scheduling for Multiple Functional Units

- Modern architectures can run operations in parallel
- List scheduling needs to be modified so that it can schedule as many operations per cycle as functional units (assuming that there are instructions available)
  - 3rd line in the list scheduling algorithm changes to:

```
while (Ready!={} &&
there_are_free_functional_units)
```

- Back to the previous example
  - assume two functional units that can issue any instruction

F. U. 1	F. U. 2	Ready Set
6. div r5,r5,100	1. load r1,@x	{6,1,4,3}
2. load r2,[r1+4]	4. mult r6,r6,100	{2,4,3}
3. and r3,r3,0x00FF	nop	{3}
nop	nop	{}
7. add r4,r2,r5	5. store r6	{7,5}
8. mult r5,r2,r4	9. store r4	{8,9}

Exercise

- Assume two functional units: one for ALU operations only and another for memory operations only. A `load` and a `mult` have a latency of 2 cycles, all other instructions have a latency of 1 cycle

```
1.  load r1, @x
2.  load r2, @y
3.  add r2,r2,42
4.  load r3, @z
5.  shl r4,r1,4
6.  store r4
7.  mult r5,r2,r3
8.  add r6,r5,r4
9.  store r5
```

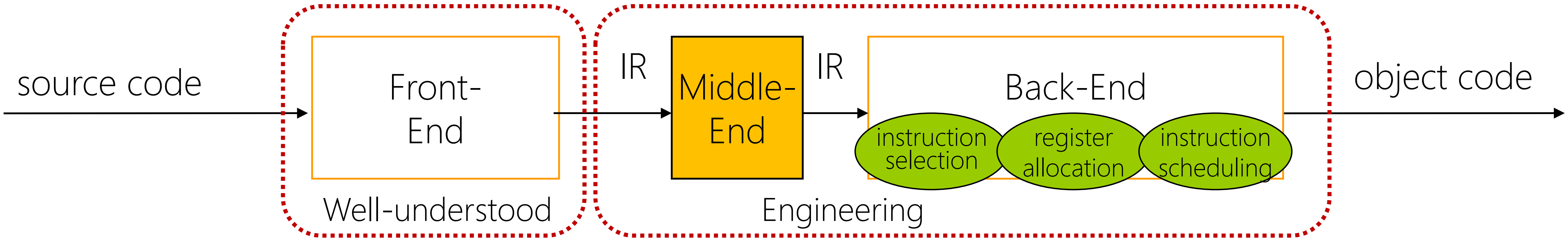
ALU	MEM
nop	2
nop	4
3	1
7	nop
5	nop
8	9
nop	6

Further Issues

- Going beyond basic blocks
  - identify high-frequency path and schedule as if a single block
- Modulo scheduling
  - schedule multiple iterations together
  - run several iterations concurrently - i.e., overlap successive iterations
- Register allocation with instruction scheduling
  - the former before the latter restricts the choices for scheduling
  - the latter before the former: if register allocation has to spill registers, the whole (carefully done) schedule changes!
- Besides performance, we may want to minimize power consumption, size of the code, …

Code Optimization

- Goal
  - improve program performance within some constraints (may also reduce size of the code, power consumption, etc…)
- Issues
  - legality: must preserve the meaning of the program
    - externally observable meaning may be sufficient/may need flexibility
  - benefit: must improve performance on average or common cases
    - predicting program performance is often non-trivial
  - compile-time cost justified: list of possible optimizations is huge
    - inter-procedural optimizations



"Optimizing" Transformations

- Finding an appropriate sequence of transformations is a major challenge: modern optimizers are structured as a series of passes
  - optimization 1 is followed by optimization 2; optimization 2 is followed by optimization 3, and so on…
- Transformations may improve program at
  - source level (algorithm specifics)
  - IR (machine-independent transformations)
  - target code (machine-dependent transformations)
- Some typical transformations
  - discover and propagate some constant value
  - remove unreachable/redundant computations
  - encode a computation in some particularly efficient form

A Classification of Compiler Optimizations

- By Scope
  - local: within a single basic block
  - peephole: on a window of instructions (usually local)
  - loop-level: on one or more loops or loop nests
  - global: for an entire procedure
  - inter-procedural: across multiple procedures or whole program (also called IPO)
  - Example: LLVM Passes
    - Module, CallGraph, Function, Loop, Region, BasicBlock
- By machine information used
  - machine-independent versus machine-dependent
- By effect on program structure
  - algebraic transformations
    - `x+0`, `x*1`, `3*z*4`, …
    - affine transformations on polyhedral loop transf.
  - reordering transformations
    - change the order of two computations
    - loop-level reordering transformations

Transformations (1)

- Common Subexpression Elimination

```
A[i][i*2+10] = B[i][i*2+10]+5;
```

→

```
tmp =  i*2+10;
A[i][tmp] = B[i][tmp]+5;
```

- Copy Propagation

```
t = i*4;
s = t;
a[s] = a[s]+4;
```

→

```
t = i*4;

a[t] = a[t]+4;
```

- Constant Propagation

```
N=64
c=2
for (i=0;i<N;i++)
  a[i] = a[i]+c;
```

→

```
N=64
c=2
for (i=0;i<64;i++)
  a[i] = a[i]+2;
```

Transformations (2)

- Constant folding

```
tmp = 5*3 + 8 - 12/2;
```

→

```
tmp = 17;
```

- Dead Code Elimination

```
if(3 > 7) {
  x = a + b;
}
```

→

```
// removed
```

- Reduction in strenght

```
x*2 + x*1024;
```

→

```
x + x + (x<<10);
```

Loop Transformations

- Very important for performance
  - change the order in which the iteration space is traversed
  - can expose parallelism; increase available ILP; improve memory behavior
- Dependence testing is required to check validity of transformation
- Automatic parallelization approaches are mainly based on loop-parallelism

Loop Merge

```
for(ia=exp1;ia<exp2;ia++)
   A(ia);
for(ib=exp1;ib<exp2;ib++)
   B(ib);
```

```
for(i=exp1;i<exp2;i++){
  A(i);
  B(i);
}
```

- Improve locality
- Reduce loop overhead

Loop Merge Example

```
for (i=0; i<N; i++)
   B[i] = f(A[i]);
for (j=0; j<N; j++)
   C[j] = f(B[j],A[j]);
```

```
for (i=0; i<N; i++){
   B[i] = f(A[i]);
   C[i] = f(B[i],A[i]);
}
```

- Dependency check
- Different memory usage, may improve locality
  - Is it always better?

Loop Body Split

```
for(i=exp1;i<exp2;i++){
  A(i);
  B(i);
}
```

```
for(ia=exp1;ia<exp2;ia++)
   A(ia);
for(ib=exp1;ib<exp2;ib++)
   B(ib);
```

Example: Loop Body Split + Merge

```
for (i=0; i<N; i++){
  A[i] = f(A[i-1]);
  B[i] = g(in[i]);
}
for (j=0; j<N; j++)
  C[i] = h(B[i],A[N]);
```

```
for (i=0; i<N; i++)
 A[i] = f(A[i-1]);
for (k=0; k<N; k++)
 B[k] = g(in[k]);
for (j=0; j<N; j++)
 C[j] = h(B[j],A[N]);
```

```
for (i=0; i<N; i++)
 A[i] = f(A[i-1]);
for (j=0; j<N; j++){
 B[j] = g(in[j]);
 C[j] = h(B[j],A[N]);
}
```

Loop Interchange

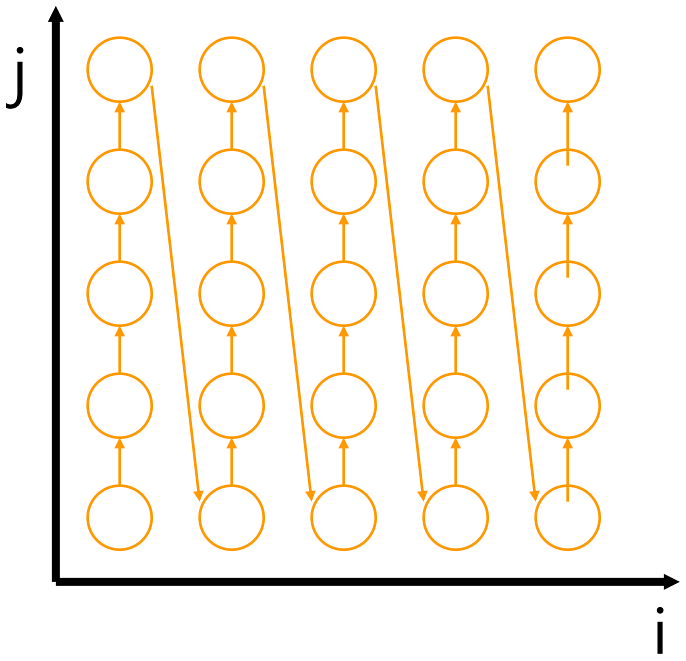
```
for(ia=exp1;ia<exp2;ia++){
  for(ib=exp3;ib<exp4;ib++){
    A(ia,ib);
```

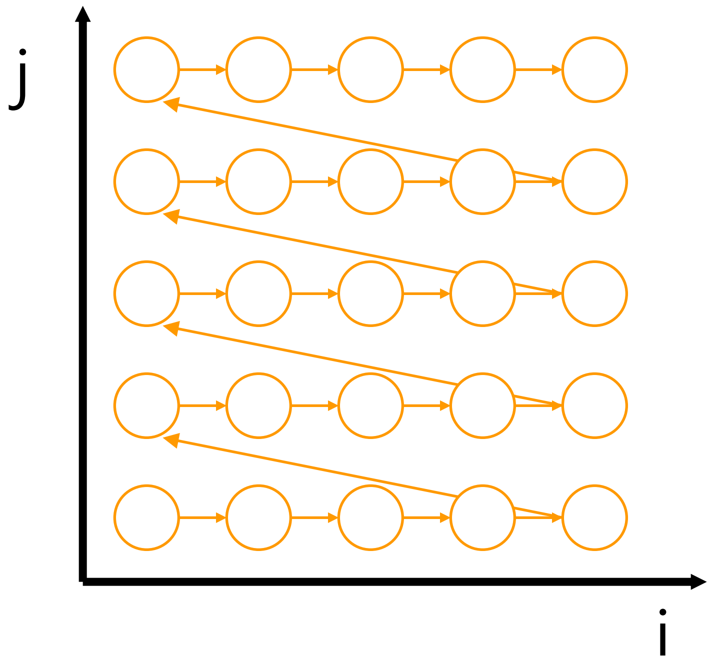
```
for(ib=exp3;ib<exp4;ib++){
  for(ia=exp1;ia<exp2;ia++){
    A(ia,ib);
```

Loop Interchange Example

```
for(j=0; j<H; j++)
  for(i=0; i<W; i++)
    A[i][j] = …;
```

```
for(i=0; i<W; i++)
  for(j=0; j<H; j++)
    A[i][j] = …;
```





Loop Tiling

Tile size

Tile factor

```
for (i=0; i<exp_1*exp_2; i++)
{
  A[i] = …;
}
```

```
for (i_1=0; i_1<exp_2; i_1++)
  for (i_2=exp_1*i_1; i_2<exp_1*(i_1+1);i_2++){
    A[i_2] = …;
}
```

- Improve cache reuse by dividing the iteration space into tiles and iterating over these tiles
- Useful when the working set does not fit into cache or when there exists much interference
  - good with parallel loop (e.g., multiple threads can work on the same data)
- Two adjacent loops can legally be tiled if they can legally be interchanged

2D Tiling Example

```
for(i=0; i<N; i++)
    for(j=0; j<N; j++)
        A[i][j] = B[j][i];
```

```
for(TI=0; TI<N; TI+=16)
  for(TJ=0; TJ<N; TJ+=16)
    for(i=TI; i<min(TI+16,N);i++)
      for(j=TJ; j<min(TJ+16,N); j++)
        A[i][j] = B[j][i];
```

- Two-dimensional case
  - from 2 to 4 loops
  - it can be further extended to N-dimensional cases
- Most tile size selection algorithms use a cache model
  - generate collection of tile sizes
  - estimate resulting cache miss rate

Loop Unrolling

```
for (i=exp1; i<exp2; i++)
  A[i]=…;
```

```
for (i=exp1/2; i<exp2/2; i++){
  A[2*i]=…;
  A[2*i+1]=…;
}
```

- Duplicate loop body and adjust loop header
- Increases available ILP, reduces loop overhead, and increases possibilities for common subexpression elimination
- Always valid

Peephole Transformation

- Optimization performed over a very small set of instructions in a segment of generated code
  - the set is called a peephole or a window
  - recognize sets of instructions that can be replaced by shorter or faster sets of instructions (replacement rules)
- Examples
  - constant folding: evaluate constant sub-expression in advance
  - strength reduction: replace slow operations with faster equivalents
  - null sequences: delete useless operations
  - combine operations: replace several operations with one equivalent
  - algebraic laws: use algebraic laws to simplify or reorder instructions
  - special case instructions: use instructions designed for special operand cases
  - address mode operations: use address modes to simplify code

Binary Translation (or Binary Recompilation)

- Binary translation is the emulation of one instruction set by another through translation of binary code
  - sequences of instructions are translated from the source to the target instruction set
  - the target instruction set may be the same as the source instruction set, providing testing and debugging features such as instruction trace, conditional breakpoints and hot spot detection
- may be implemented
  - as peephole transformation
  - with instruction lifting
    - e.g., from x86 to LLVM IR (bitcode)
- Static binary translation
  - difficult: not all code can be discovered by a translator, e.g., indirect branches (value known at runtime)
- Dynamic binary translation
  - apply to a short sequence of code (e.g., basic block)
  - when possible, and branch instructions are made to point to already translated and saved code (memoization)

Binary Translation: Examples

- Static
  - x86 to ARM translator
  - x86 to x64
- Dynamic
  - QuickTransit (Apple/Transitive): SPARC to x86, x86 to Power architecture, MIPS to Itanium, PPC to x86
  - Intel : from IA-32 to Itanium
- Hardware binary translation
  - x86 Intel CPUs since the Pentium Pro translate complex CISC x86 instructions to more RISC-like internal micro-operations