TUHH
Hamburg University of Technology

Institute of
Embedded Systems

Compilers for Embedded Systems

Summer Term 2022

Heiko Falk

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

Chapter 6

Code Generation

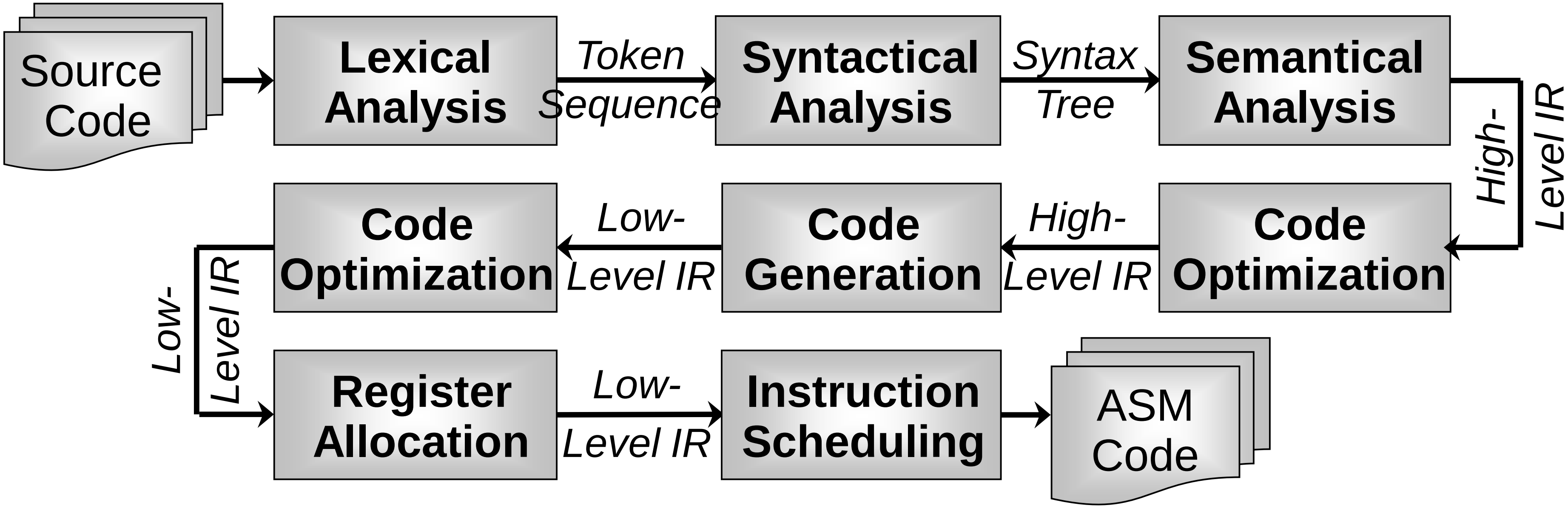
Outline

Chapter Contents

6. Code Generation

- Introduction
    - Role of Code Generation
    - Data Flow Graphs
    - Code Generator Generators
- Tree Covers using Dynamic Programming
    - Partitioning of Data Flow Graphs into Data Flow Trees
    - Tree Covering
    - Tree Pattern Matching Algorithm
    - Tree Grammars for Rule-based Derivation of Code
- Discussion

Role of the Instruction Selection



Code Generation

- Selection of machine instructions in order to implement an IR
- “Heart” of a compiler that performs the actual translation of source into target language

Goals

Synonyms

- “Code Generation”, “Instruction Selection” and “Code Selection” are often used synonymously

Inputs and Outputs

- Input: An intermediate representation *IR* to be translated
- Output: A Program *P(IR)*
  (often in assembly or machine code, but often also another IR)

Requirements

- *P(IR)* must be semantically equivalent to *IR*
- *P(IR)* must be efficient regarding an objective function

Data Flow Graphs

What does “semantically equivalent to *IR*” mean...?

- *P(IR)* must have a data flow that is equivalent to that of *IR*, under consideration of control flow dependencies.

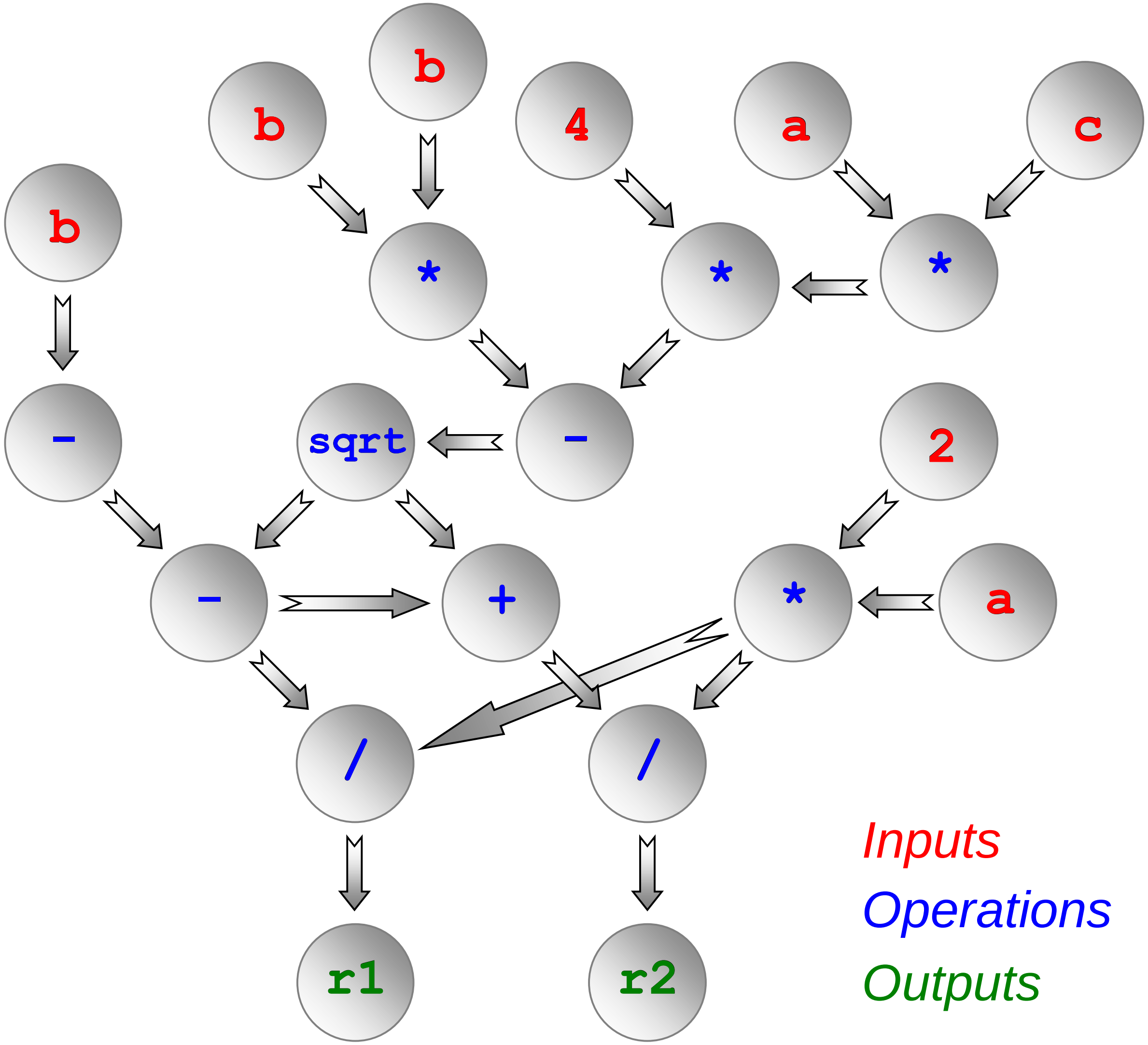
Definition (Data Flow Graph):

Let $B = (I_1, ..., I_n)$ be a basic block *(☞ Chapter 3)*. The *Data Flow Graph (DFG)* of B is a directed, acyclic graph $DFG = (V, E)$ with

- each node $v \in V$ represents either
  - an input value for B (input variable, constant)
  - or a single operation within $I_1, ..., I_n$
  - or an output value of B
- edge $e = (v_i, v_j) \in E \Leftrightarrow v_j$ uses data that v_i computes

Example

```
t1 = a * c;
t2 = 4 * t1;
t3 = b * b;
t4 = t3 - t2;
t5 = sqrt( t4 );
t6 = -b;
t7 = t6 - t5;
t8 = t7 + t5;
t9 = 2 * a;
r1 = t7 / t9;
r2 = t8 / t9;
```



Code Generation

Problem Formulation

- To cover all nodes of all DFGs of *IR* with semantically equivalent operations of the target language

Implementation of a Code Generator

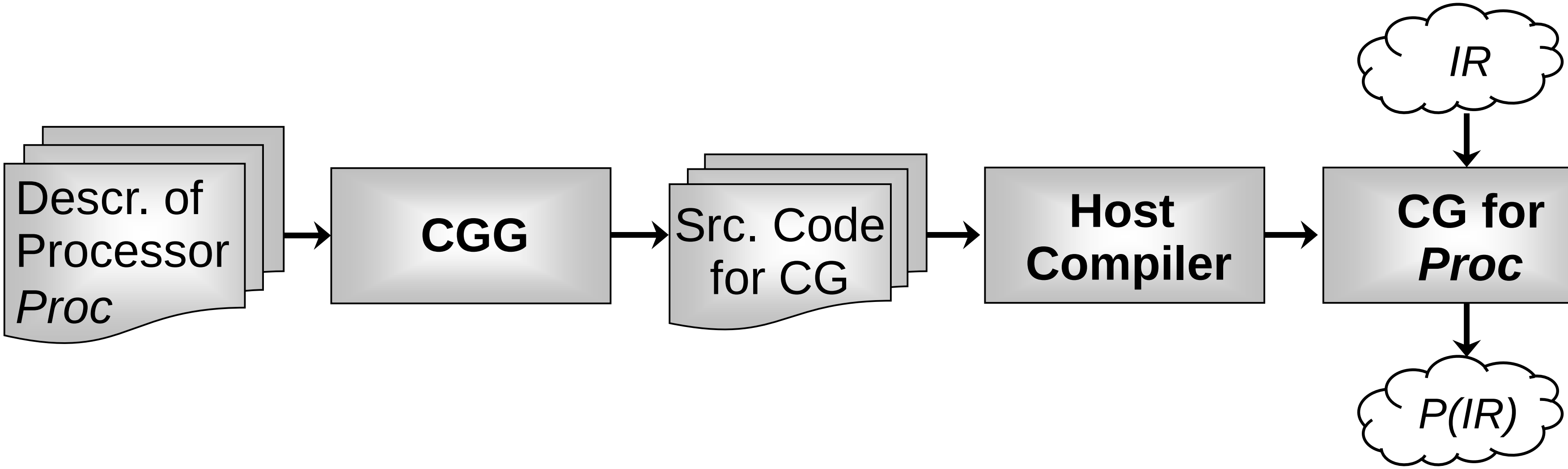
- Non-trivial task, highly dependent of the target processor’s architecture
- Manual implementation of a code generator not affordable for today’s processors’ complexity

☞ Instead: Use of so-called *Code Generator Generators*

Code Generator Generators

Workflow

- So-called *Meta Programs*, i.e. programs that produce other programs as output.
- A code generator generator *(CGG)* receives a processor description as input and generates a code generator *(CG)* from it for exactly that processor



Chapter Contents

Tree Pattern Matching (TPM)

Motivation

– 3-SAT polynomially reducible to covering of data flow *graphs*

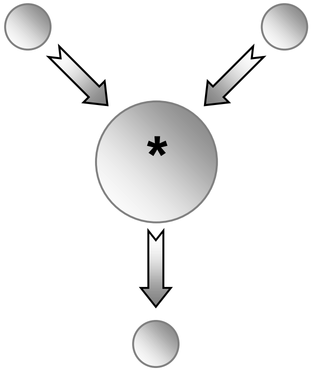
> *[**J. Bruno, R. Sethi.** Code generation for a one-register machine. Journal of the ACM 23(3), Jul 1976]*

☞ Optimal instruction selection is thus NP complete

– ***But:*** Machine operations of typical processors usually have tree-like data flow

☞ ***Tree-based Code Generation***

– Optimal tree-based instruction selection feasible efficiently, in polynomial run-time



Workflow of Tree Pattern Matching

Given

- An intermediate representation *IR* to be translated

Approach

- Program $P = \emptyset$;
- For each basic block *B* from *IR*:
  - Determine data flow graph *D* of *B*
  - Partition *D* into single data flow trees (DFT) $T_1, ..., T_N$
  - For each DFT T_i:
    - $P = P \cup \{$ Optimal code from Tree-Covering of $T_i \}$
- Return *P*

Partitioning of DFGs into DFTs

Definition (Common Subexpression):

Let *DFG* = (*V*, *E*) be a data flow graph.
A node $v \in V$ with more than one outgoing edge is called *common subexpression (CSE)*.

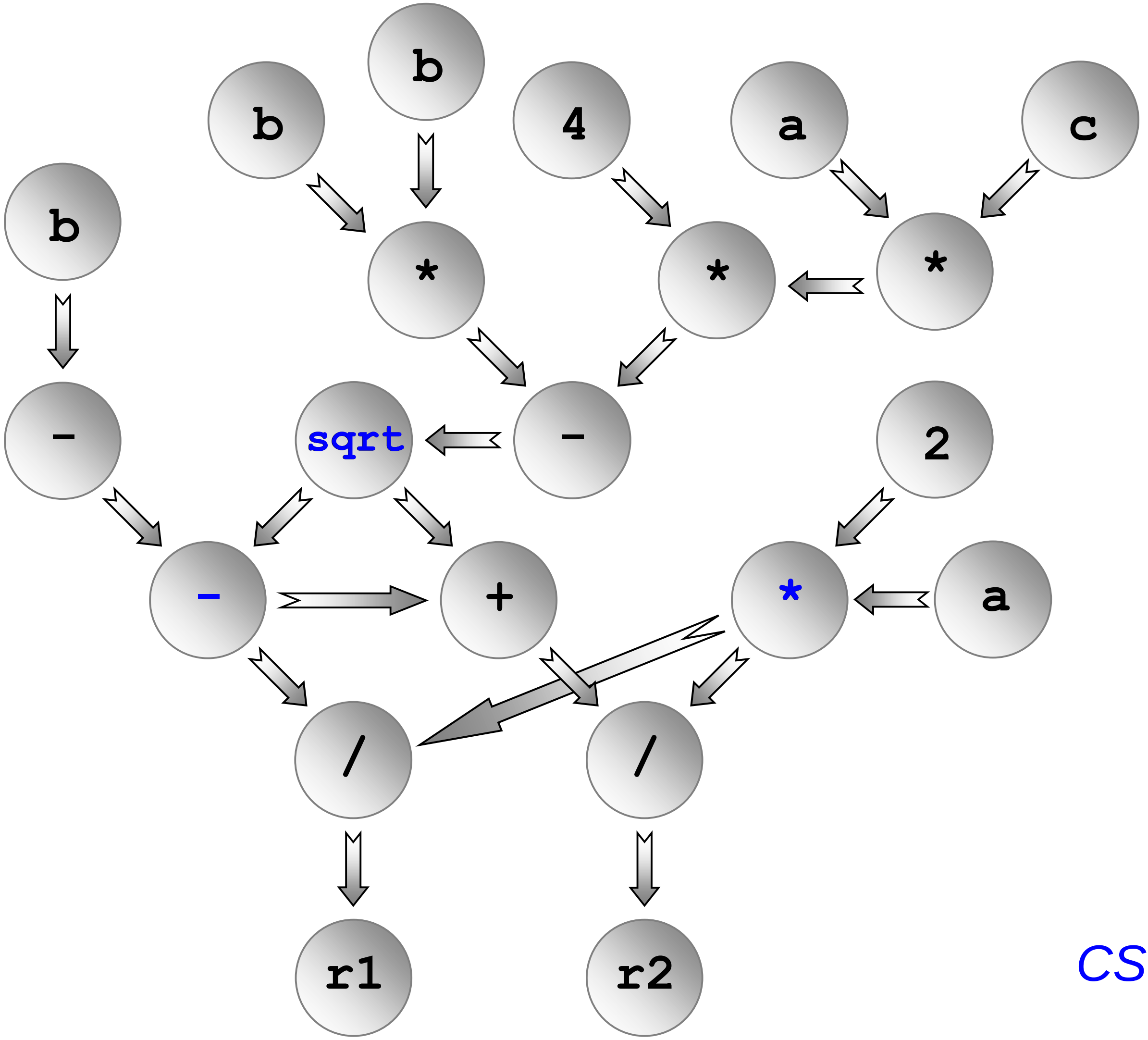
Definition (Data Flow Tree):

A data flow graph *DFG* = (*V*, *E*) without any CSE is called *data flow tree (DFT)*.

☞ **DFG Partitioning**

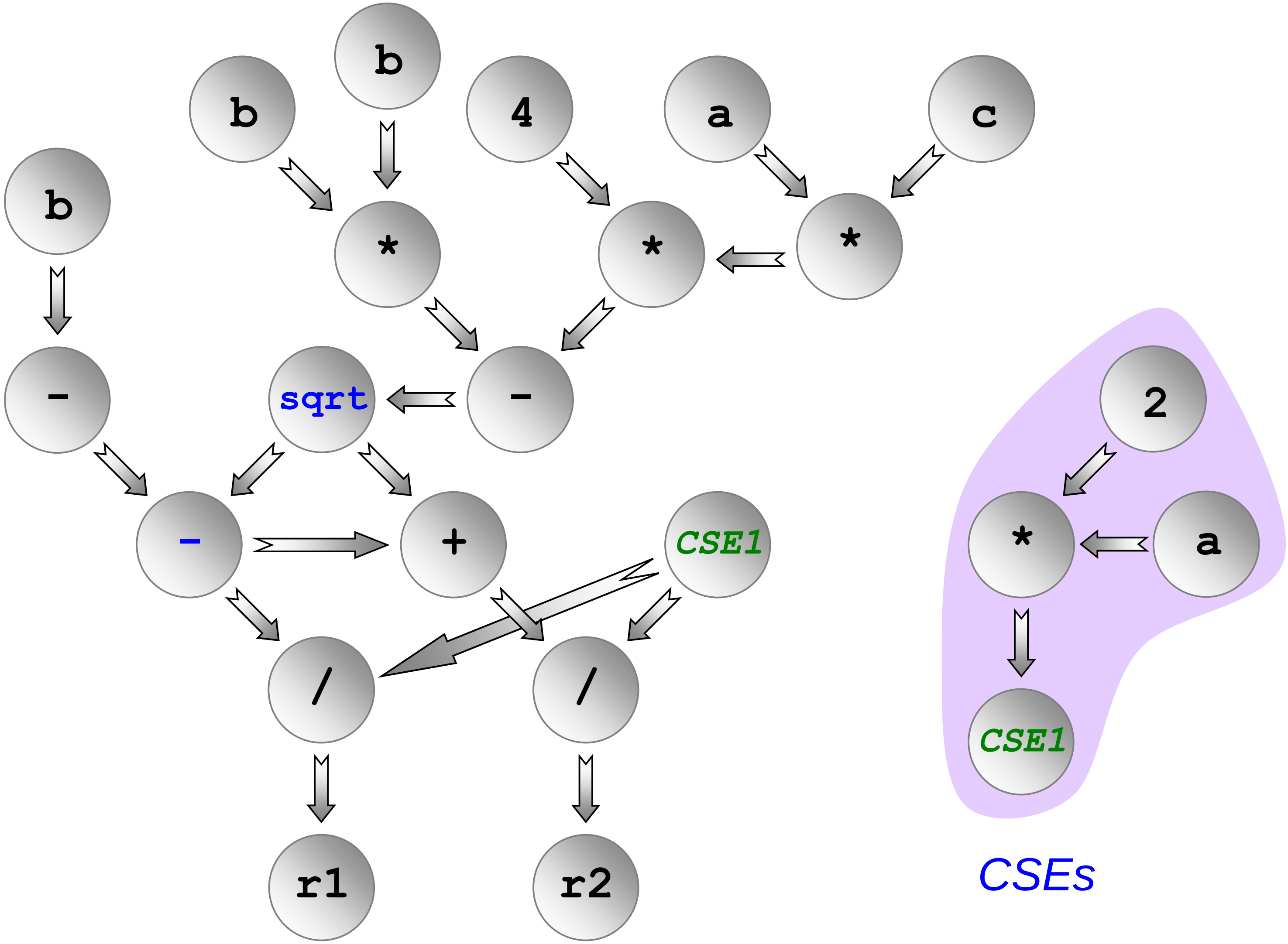
- Splitting of the DFG into DFTs along the contained CSEs
- For each CSE: Add intermediate nodes to the resulting trees

Example (1)



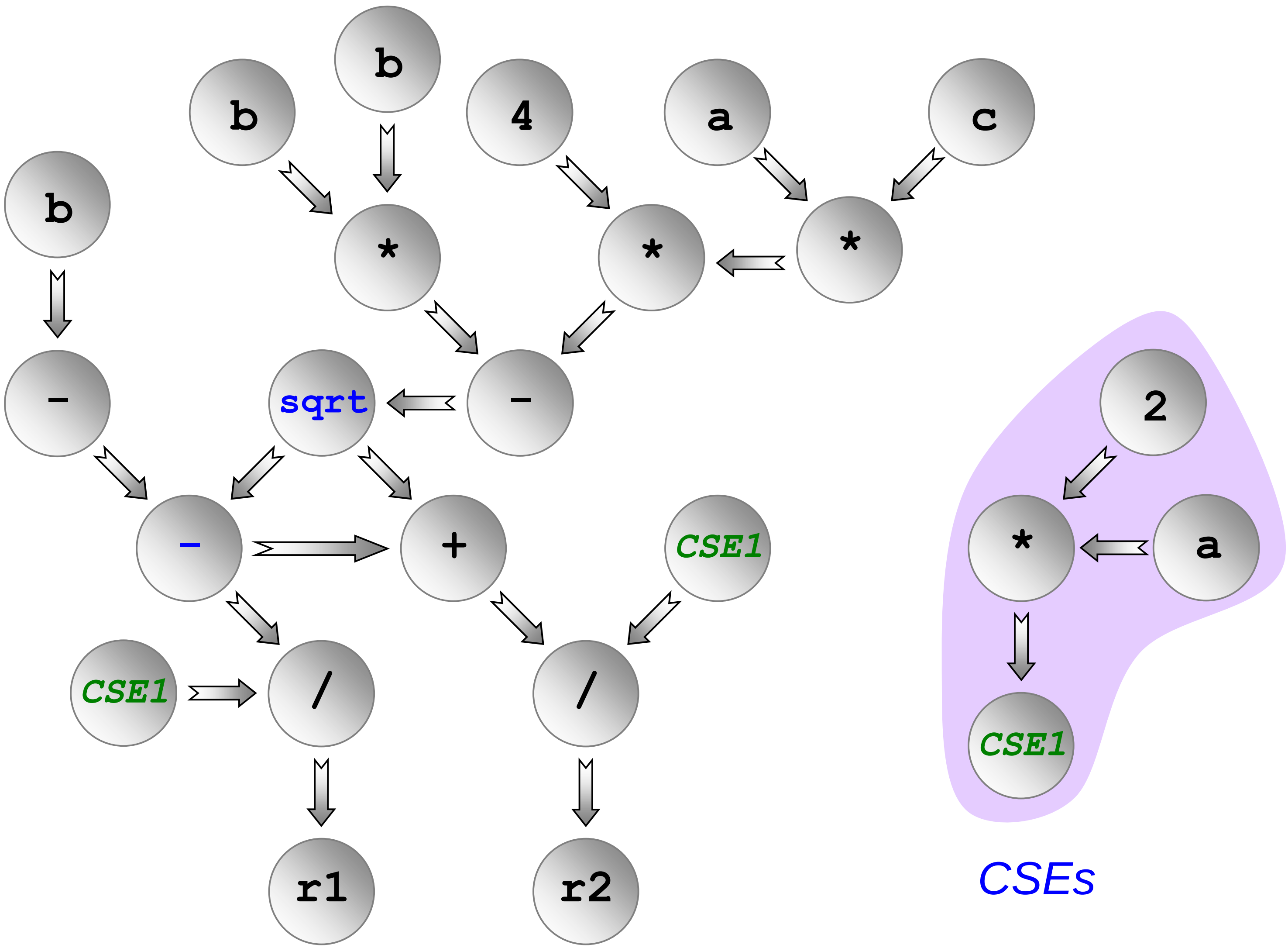
CSEs

Example (2)



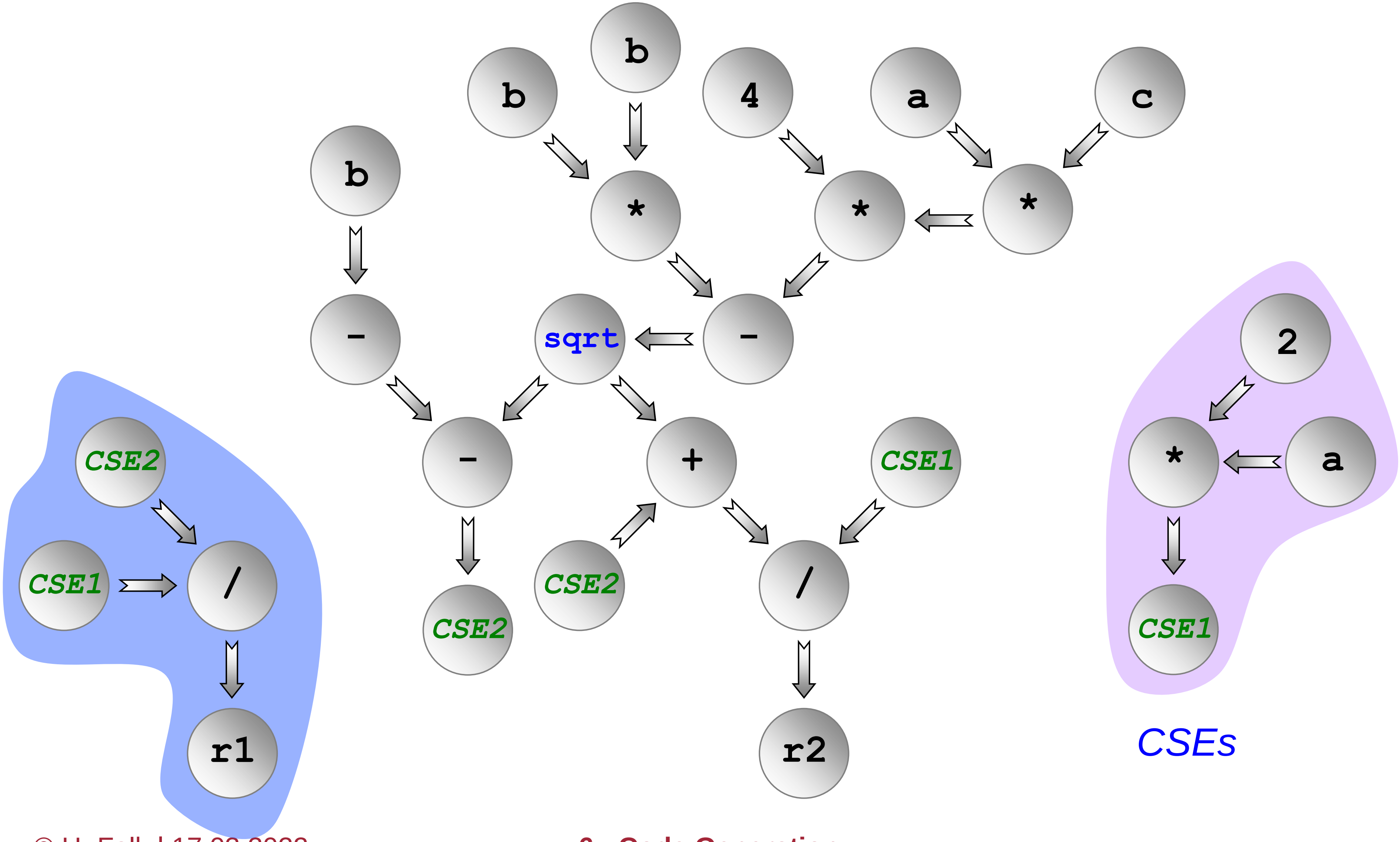
CSEs

Example (3)

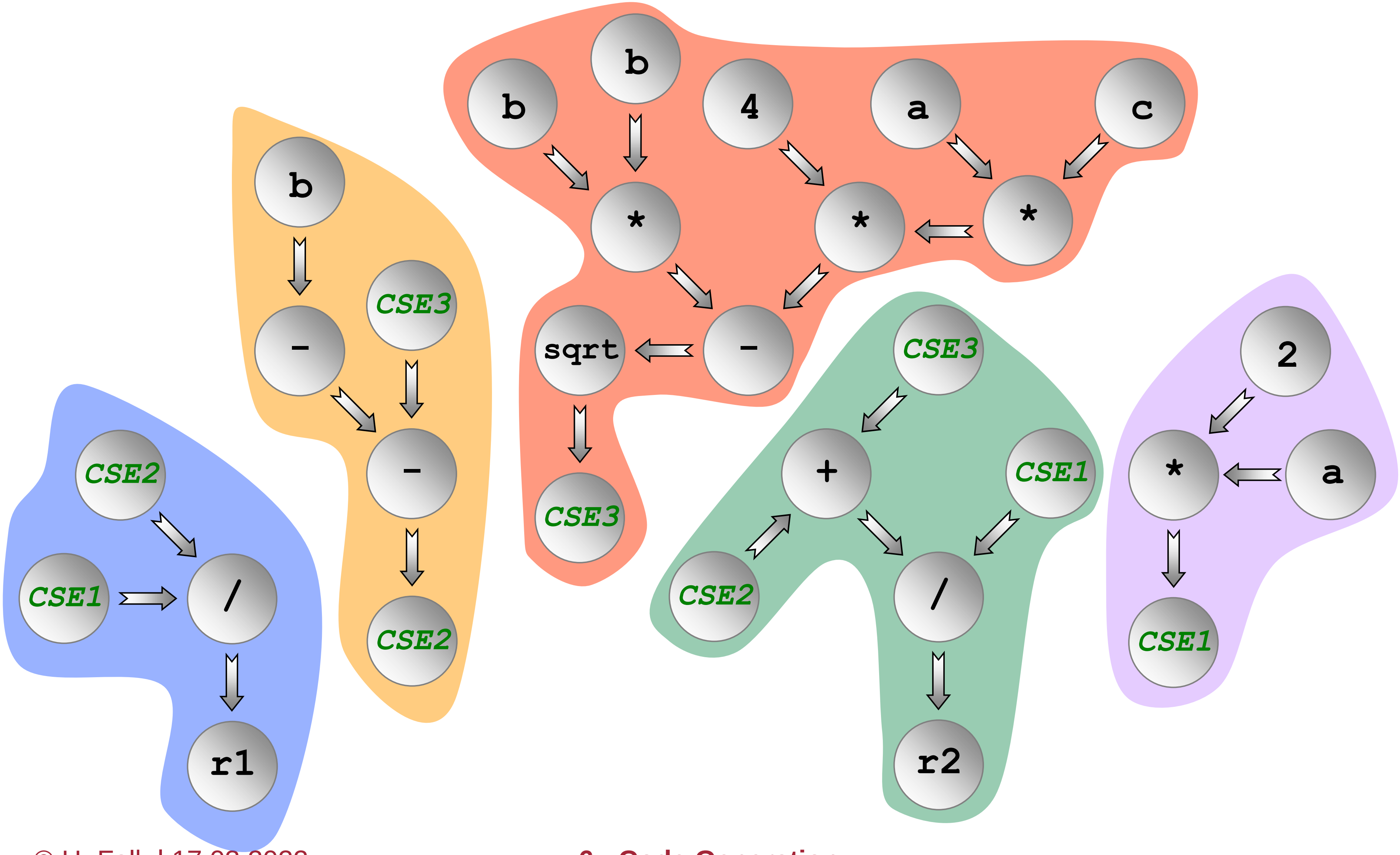


CSEs

Example (4)



Example (5)


b
b
b
4
a
c
*
*
*
-
sqrt
CSE3
CSE3
-
-
CSE2
CSE3
+
CSE1
CSE2
/
r2
2
*
a
CSE1
CSE2
CSE1
/
r1

Tree Cover

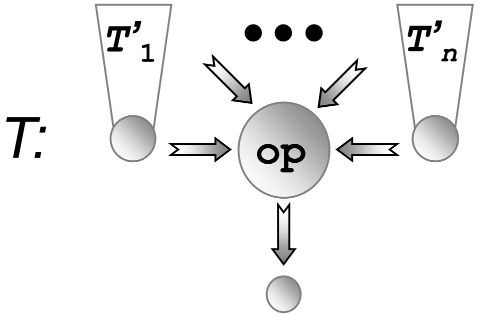
Definition (Tree-Covering by an Operation Sequence):

Let $T = (V, E)$ be a DFT, $S = (o_1, ..., o_N)$ be a sequence of machine operations. The last operation o_N shall have the format $\mathtt{d} \leftarrow \mathtt{op}(\mathtt{s}_1, ..., \mathtt{s}_n)$.

Let $S'_1, ..., S'_n$ denote those sub-sequences of S that compute the operands $\mathtt{s}_1, ..., \mathtt{s}_n$ of o_N, respectively.

S covers T iff

- the operator `op` corresponds to the root of T, i.e., T can be depicted as follows:



- and if each S'_i by itself also covers T'_i, respectively ($1 \leq i \leq n$).

Examples for Covers

TriCore Instruction Set:

`add  Dc, Da, Db` *(Dc = Da + Db)*

`mul  Dc, Da, Db` *(Dc = Da * Db)*

`madd Dc, Dd, Da, Db` *(Dc = Dd + Da * Db)*

☞ Operation **`add %d4, %d8, %d9`** covers *T1*

☞ Operation **`mul %d10, %d11, %d12`** covers *T2*

☞ *Also evident:* Operation sequence

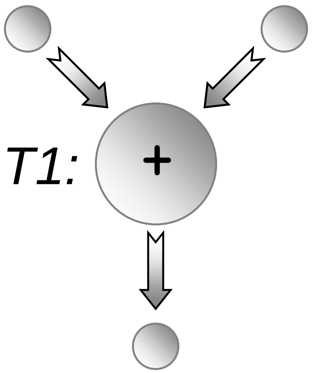
`mul %d10, %d11, %d12`

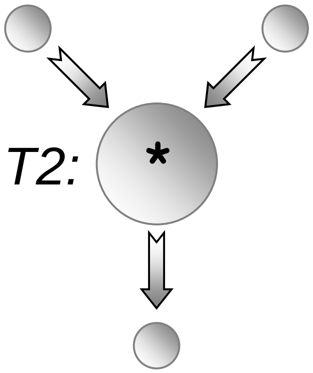
`add %d4, %d8, %d10` covers *T3*

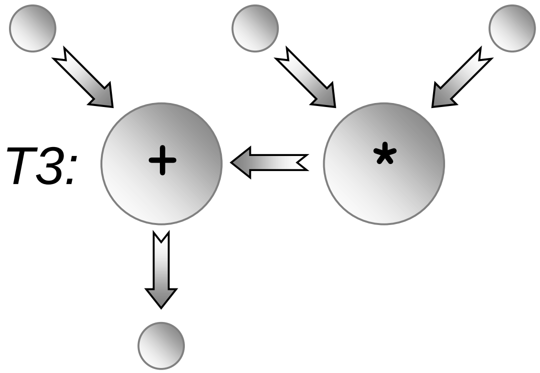
☞ *Additionally:* Singleton sequence

`madd %d4, %d8, %d11, %d12` also covers *T3*.

Data Flow Trees:







Tree Pattern Matching Algorithm (1)

Given

- DFT $T = (V, E)$, let node $v_0 \in V$ be output (i.e. root) of T
- Set O of all machine operations o of the target processor's instruction set
- Cost function c: $O \rightarrow \mathbb{N}$ (e.g., size of each operation in bytes)
- Number $K \in \mathbb{N}$ of all registers of the target processor

Data Structures

- Array $C[j][v]$: Holds the minimal costs per node $v \in V$ according to cost function c, if a total of j registers is available to compute that sub-tree of T with root v.
- Array $M[j][v]$: Holds the cost-optimal machine operation from O and the optimal operand order per node $v \in V$, if a total of j registers is available.

Tree Pattern Matching Algorithm (2)

Workflow – `TPM( DFT` *T* `):`

- initialize(*T*);
- computeCosts(*T*);
- generateCode(*T*, *K*);

Phase 1 – `initialize( DFT` *T* `):`

- For all possible register numbers $1 \leq j \leq K$ and for all nodes $v \in V$:

$$C[j][v] = \begin{cases} 0 & \text{if } v \text{ is input/leaf of } T \\ \infty & \text{otherwise} \end{cases}$$

- For all possible register numbers $1 \leq j \leq K$ and for all nodes $v \in V$:

$$M[j][v] = (\emptyset, \emptyset)$$

Tree Pattern Matching Algorithm (3)

Phase 2 – `computeCosts( DFT T ):`

- For all nodes $v \in V$ in post-order sequence starting at root node v_0:
  - Let T' be that sub-tree of T with current node v as root
  - For all operations $o \in O$ that cover v:
    - Use o and partition T' into sub-trees $T'_1, \ldots, T'_n$ with roots $v'_1, \ldots, v'_n$, respectively, according to Tree Cover-Definition (☞ Slide 20)
    - For each $1 \leq j \leq K$ and all permutations π over $(1, \ldots, n)$:
      - Compute the minimal costs for node v:

$$C[j][v] = \min(C[j][v], \sum_{i=1}^{n} C[j-i+1][v'_{\pi(i)}] + c(o))$$

$M[j][v] =$ That pair (o, π) that leads to the minimal costs $C[j][v]$

Tree Pattern Matching Algorithm (4)

Phase 3 – `generateCode(` *`DFT`* T`,` `int` j `):`

- Let $v \in V$ be the root of T
- Operation o = first element of $M[j][v]$
- Permutation π = second element of $M[j][v]$
- Use o and partition T into sub-trees $T_1, \ldots, T_n$ according to Tree Cover-Definition
- For each $i = 1, \ldots, n$: **`generateCode(`** $T_{\pi(i)}$, $j - i + 1$ **`)`**
- Generate machine code for operation o

*[**A. Aho, S. Johnson.** Optimal Code Generation for Expression Trees. Journal of the ACM 23(3), Jul 1976]*

Remarks (1)

- *Post-order traversal:* For the root v of T, visit at first the children $v_1, ..., v_n$ in post-order sequence, then finally visit v itself.
- *Permutation* π*:* For the current node v and sub-trees $T'_1, ..., T'_n$ with roots $v'_1, ..., v'_n$, a permutation π describes one possible order in which the sub-trees can be evaluated.
  E.g., $\pi = (2, 3, 1)$ states that sub-tree 2 is evaluated first, then sub-tree 3, and finally sub-tree 1.
- **`computeCosts`** computes the minimal costs for each node v under consideration of *all* possible evaluation orders of v's children (i.e., all permutations π) and *all* possible amounts of free registers (i.e., all values $j \in [1, K]$).

Remarks (2)

- For each tree T, the TPM algorithm always tracks how many of the K registers of the processor are still free, i.e., it does not work with an infinite amount of available virtual registers.
- Accordingly, costs are computed in dependence of the number j of available registers.
- For some nodes $v'_1, \ldots, v'_n$ and a given value of j, the costs can vary, depending on the permutation π!

Remarks (3)

Example: Assume that *j* = 3 registers are available to evaluate the current node *v*. The evaluation of sub-tree T'_1 requires 2 free registers, but that of T'_2 3 registers.

- $\pi = (1, 2)$: If T'_1 were evaluated first, 2 registers are occupied meanwhile and the result of T'_1 is stored permanently in one of the 3 free registers afterwards. Thus, only 2 registers are available to evaluate T'_2.
  But since T'_2 requires 3 registers, additional memory transfer instructions need to be generated to evaluate T'_2 which consequently increase costs.
- $\pi = (2, 1)$: During its evaluation, T'_2 occupies all 3 available registers, the result of T'_2 is permanently stored in one if the 3 free registers afterwards. Thus, only 2 registers are available to evaluate T'_1. But since T'_1 only needs 2 registers for its evaluation, no additional memory transfer instructions are required, thus leading to minimal costs.

Run-Time Complexity of TPM

Assumptions

- A processor's instruction set is given and is fixed
- The size of the set O of machine operations is constant
- The number of possible permutations π is constant, too, since the number of operands per machine operation in the instruction set is also constant
  (typically 2 or 3 operands per operation)

Cost Computation

- Since the algorithm's loops over all machine operations $o \in O$ and over all permutations π only contribute a constant factor:
- Linear complexity in terms of the size of T: $O(|V|)$

Code Generation

- Obviously, also linear complexity in terms of the size of T: $O(|V|)$

Open Issues

TPM Algorithm as presented here formulated generically.

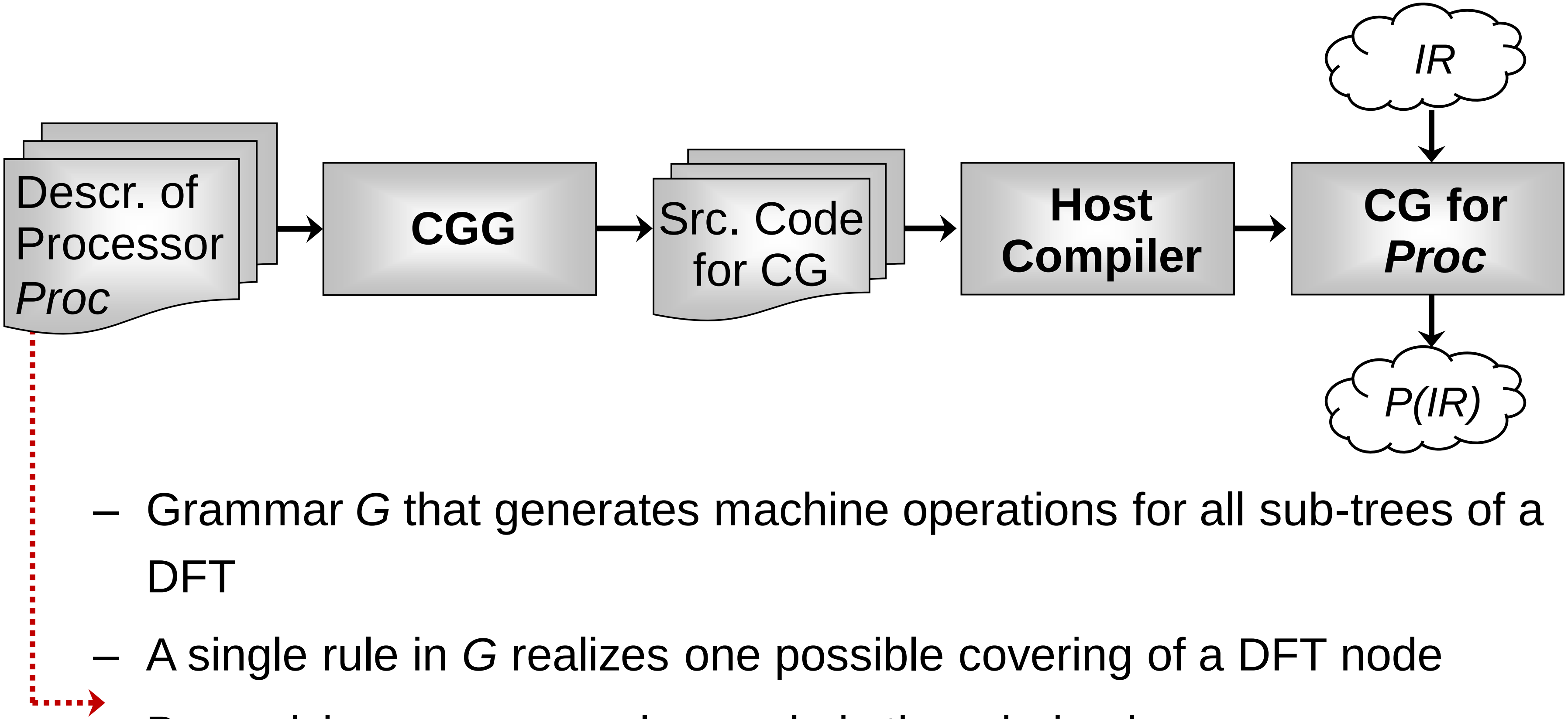
☞ ***How is this algorithm adapted for some actual processor architecture?***

Details to be clarified

- How is the matching of a machine operation `op` with the root of T realized (cf. Tree Cover-Definition)?
- In which format are the set O of all machine operations and the cost function C specified for the TPM algorithm?
- How does TPM handle the storage of the optimal machine operation o in M and how is the actual code generation for o done?

In the following: Assumption of an infinite amount of virtual registers

Processor Description per Tree Grammar



- Grammar *G* that generates machine operations for all sub-trees of a DFT
- A single rule in *G* realizes one possible covering of a DFT node
- By applying grammar rules, code is thus derived
- Each individual derivation/rule produces costs

Structure of a Tree Grammar (1)

Based on Code Generator Generator *icd-cg*:

- Tree grammar G consists of rules R_1, ..., R_r
- Each rule R_i has a signature consisting of terminal and non-terminal symbols:

  $<\text{nonterminal}_{i,0}>$`:` **`<terminal`**$_{i,1}$**`>`**`(` $<\text{nonterminal}_{i,2}>$`, …,` $<\text{nonterminal}_{i,n}>$ `)`

  (Specification of non-terminals in (...) optional)

  (So-called chain-rules $<\text{nonterminal}_{i,0}>$`:` $<\text{nonterminal}_{i,1}>$ *also valid)*
- Terminals: Possible nodes in a DFT *T*

  (e.g., **`tpm_BinaryExpPLUS`**, **`tpm_BinaryExpMULT`**, *... in ICD-C)*
- Non-terminals: Usually processor-specific classes of memories where source and target operands of operations can be stored

  (e.g., data & address registers, constant immediate values, ...)

Structure of a Tree Grammar (2)

Example *(based on ICD-C & TriCore 1.3)*

- Rule

  `dreg: tpm_BinaryExpPLUS( dreg, dreg )`

  responsible to cover the binary operator **+** of ANSI-C where both summands reside in data registers and the sum also stays in a data register.
- Rule

  `dreg: tpm_BinaryExpMULT( dreg, const9 )`

  responsible to cover the binary operator ***** of ANSI-C with the first factor in a data register, the second one given as signed 9-bit immediate value, and the product residing in a data register.

Structure of a Tree Grammar (3)

Based on Code Generator Generator *icd-cg*: *(ctd.)*

- Terminal and non-terminal symbols must be declared in tree grammar *G*.
- Overall structure of a file for a tree grammar *G*:

```
%{          // Preamble
%}

%term <terminal_1>
%term <terminal_2>
...

%declare <nonterminal_0>;
%declare <nonterminal_1>;
...

%%

Rule_1;
Rule_2;
...

%%
```

Structure of a Tree Grammar (4)

Based on Code Generator Generator *icd-cg*: *(ctd.)*

- The specification of each rule R_i of the tree grammar consists of signature, cost part and action part:

```
<nonterminal_{i,0}>: <terminal_{i,1}>( <nonterminal_{i,2}>, …,
                                       <nonterminal_{i,n}> )
{
  // Code for cost computation
}
=
{
  // Code for action part
};
```

Structure of a Tree Grammar (5)

Based on Code Generator Generator *icd-cg*: *(ctd.)*

- Cost part of R_i assigns costs to nonterminal$_{i,0}$ that arise if R_i is used to cover `terminal`$_{i,1}$.
- Cost part can contain any arbitrary, user-specified C/C++-Code for cost computation.
- Costs can represent, e.g., the number of generated machine operations, code size, ...
- Costs of a rule R_i can be set to ∞ explicitly if R_i shall not be used at all for a tree cover in particular situations.
- C/C++ data type for costs, a feasible “less than” comparison operator, and values for zero and infinite costs need to be declared in the preamble of G.

Structure of a Tree Grammar (6)

Example *(based on ICD-C & TriCore 1.3)*

```
// Preamble
typedef int COST;
#define DEFAULT_COST 0;
#define COST_LESS(x, y) ( x < y )
COST COST_INFINITY = INT_MAX;
COST COST_ZERO = 0;
...
```

- Declaration of a simple cost measure – identically with `int` here
- Comparison of costs using **<** operator for `int`
- Default, zero and **∞** costs set to 0 and maximal `int` value, resp.

Structure of a Tree Grammar (7)

Example *(based on ICD-C & TriCore 1.3)*

```
dreg: tpm_BinaryExpPLUS( dreg, dreg )
{
  $cost[0] = $cost[2] + $cost[3] + 1;
} = {};
```

- Use of the pre-defined keyword **`$cost[j]`** to access the costs of nonterminal$_{i,j}$
- Costs of binary **+** with both summands in data registers (**`$cost[0]`**) are equal to costs for the first summand (**`$cost[2]`**) plus costs for the second summand (**`$cost[3]`**), plus one additional operation (**`ADD`**)

Structure of a Tree Grammar (8)

Based on Code Generator Generator *icd-cg*: *(ctd.)*

- Action part of R_i is executed if R_i is that rule with the minimal costs that covers `terminal`$_{i,1}$.
- Action part can contain any arbitrary, user-specified C/C++-Code for code generation.
- Use of the pre-defined keyword `$action[j]` to execute the action part of an operand nonterminal$_{i,j}$
- Non-terminal symbols can be declared in G to have parameters and return values in order to pass information between action parts of different rules.

Structure of a Tree Grammar (9)

Example ***(based on ICD-C & TriCore 1.3)***

```
dreg: tpm_BinaryExpPLUS( dreg, dreg ) {}={
  if (target.empty()) target = getNewRegister();
  string r1($action[2]("")), r2($action[3](""));
  cout << "ADD " << target << ", " << r1
       << ", " << r2 << endl;
  return target;
};
```

- First, determine register where to store the target operand
- Next, invocation of code generation for both source operands via `$action[2]()` and `$action[3]()`
- Finally, code generation for the addition itself

Structure of a Tree Grammar (10)

Example ***(based on ICD-C & TriCore 1.3)***

```
dreg: tpm_BinaryExpPLUS( dreg, dreg ) {}={
  if (target.empty()) target = getNewRegister();
  string r1($action[2]("")), r2($action[3](""));
  cout << "ADD " << target << ", " << r1
       << ", " << r2 << endl;
  return target;
};
```

- In order to generate code for the `ADD` operation, the above rule must know in which data registers the two summands actually reside.
- The code for the summands is, however, produced by some completely different rules of the grammar.

Structure of a Tree Grammar (11)

Example ***(based on ICD-C & TriCore 1.3)***

```
dreg: tpm_BinaryExpPLUS( dreg, dreg ) {}={
  if (target.empty()) target = getNewRegister();
  string r1($action[2]("")), r2($action[3](""));
  cout << "ADD " << target << ", " << r1
       << ", " << r2 << endl;
  return target;
};
```

– The action parts of the summands’ rules return the name of exactly this register (here naively as `string`) as result after their respective invocation via `$action[2]()` or `$action[3]()`.

Structure of a Tree Grammar (12)

Example ***(based on ICD-C & TriCore 1.3)***

```
%declare<string> dreg<string target>;
```

- Declaration of a non-terminal symbol for virtual data registers
- An action part can return a **`string`** that denotes that data register in which the action part has actually stored its target operand.
- A **`string`** can be passed as parameter **`target`** to action parts of rules producing a dreg in order to force these action parts to use a specific data register where the target operand shall be stored.

Structure of a Tree Grammar (13)

Example ***(based on ICD-C & TriCore 1.3)***

```
dreg: tpm_BinaryExpPLUS( dreg, dreg ) {}={
  if (target.empty()) target = getNewRegister();
  $action[2]("D15");
  string r2($action[3](""));
  cout << "ADD " << target << ", D15, " << r2 << endl;
  return target;
};
```

– The above rule generates a specialized variant of the TriCore's addition where the first summand must mandatorily reside in data register **`D15`**.

Tree Covers and Tree Grammars

Tree Covers

A rule R_i from G with signature

<nonterminal$_{i,0}$>`:` **`<terminal`**$_{i,1}$**`>`**`(` <nonterminal$_{i,2}$>`,` …`,`
<nonterminal$_{i,n}$> `)`

covers a DFT T iff

- the terminal symbol of R_i corresponds to the current DFT node, and
- the costs of R_i if applied to the current DFT node are less than **∞**, and
- there are rules in G that by themselves cover sub-tree T'_j and that produce a non-terminal symbol of class <nonterminal$_{i,j}$>, respectively ($2 \leq j \leq n$).

TPM Algorithm for Tree Grammars

Phase 1 – Initialization: Unchanged

Phase 2 – Cost Computation:

– Instead of determining all operations $o \in O$ that cover sub-trees T':

☞ Determine set R' of all rules $R_i \in G$ that cover T'

☞ Compute $C[v]$ as before, but now only by executing the code of the cost parts of all rules from R'

☞ Store that rule $R^{opt} \in R'$ with minimal costs $C[v]$ in $M[v]$

Phase 3 – Code Generation:

– For the root $v_0 \in T$: Invoke action part of the optimal rule $M[v_0]$

– `$action[]` calls embedded in another rule's action parts always refer to the action parts of the cost-optimal rule R^{opt}

Complex Example (1)

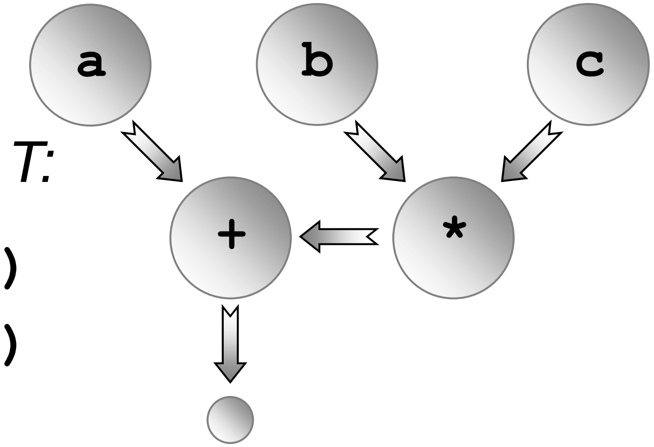
```
dreg: tpm_BinaryExpPLUS( dreg, dreg ) {
  $cost[0] = $cost[2] + $cost[3] + 1;
}={
  if ( target.empty() ) target = getNewRegister();
  string r1( $action[2]("") ), r2( $action[3]("") );
  cout << "ADD " << target << "," << r1 << "," << r2 <<
  endl; return target;
};

dreg: tpm_BinaryExpMULT( dreg, dreg ) {
  $cost[0] = $cost[2] + $cost[3] + 1;
}={
  if ( target.empty() ) target = getNewRegister();
  string r1( $action[2]("") ), r2( $action[3]("") );
  cout << "MUL " << target << "," << r1 << "," << r2 <<
  endl; return target;
};
```

Complex Example (2)

```
dreg: tpm_SymbolEXP {
  $cost[0] = $1->getExp()->getSymbol().isGlobal() ?
    COST_INFINITY : COST_ZERO;
}={
  target = "r_" + $1->getExp()->getSymbol().getName();
  return target;
};
```

- Rule assigns a virtual register to local variables used inside DFT *T*
- `$1` is the node of the DFT *T* that is to be covered by the terminal symbol
- `$1->getExp()->getSymbol()` returns the symbol / the local variable of the IR
- In case of a global variable, this rule produces costs ∞ so that it is not used
- For local variables: Costs 0 since no actual code is generated

Complex Example (3)



- C snippet **`a + (b * c)`** with DFT *T* is covered by rules

  dreg: **`tpm_SymbolExp`**

  dreg: **`tpm_BinaryExpPLUS`**`(` dreg`,` dreg `)`

  dreg: **`tpm_BinaryExpMULT`**`(` dreg`,` dreg `)`

- Costs for *T*:

  C[**+**] = *C*[**a**] + *C*[*****] + 1 = *C*[**a**] + (*C*[**b**] + *C*[**c**] + 1) + 1 = 2

- Code generated for *T*:

```
MUL r_0, r_b, r_c
ADD r_1, r_a, r_0
```

Complex Example (4)

```
typedef pair<string, string> regpair;
%declare<regpair> virtmul;

virtmul: tpm_BinaryExpMULT( dreg, dreg ) {
  $cost[0] = $cost[2] + $cost[3];
}={
  string r1( $action[2]("") ), r2( $action[3]("") );
  return make_pair( r1, r2 );
};
```

- Novel non-terminal virtmul represents a multiplication in *T* for which, however, no code shall be generated directly by a rule.
- Instead, a rule producing virtmul simply returns a pair of registers that stores where the operands of the multiplication reside.
- For lack of generated code: $C[v]$ = Sum of the costs of the operands

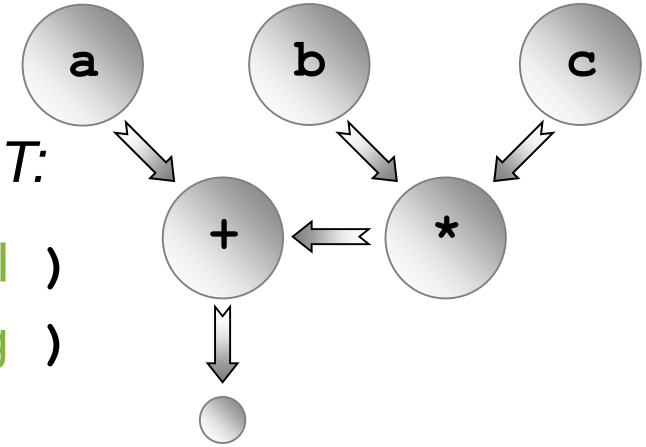
Complex Example (5)

```
dreg: tpm_BinaryExpPLUS( dreg, virtmul ) {
  $cost[0] = $cost[2] + $cost[3] + 1;
}={
  if ( target.empty() ) target = getNewRegister();
  string r1( $action[2]("") );
  regpair rp( $action[3]() );
  cout << "MADD " << target << "," << r1 << ","
       << rp.first << "," << rp.second << endl;
  return target;
};
```

- This rule becomes active, i.e., can be used to cover a tree, if the second summand is such a virtual multiplication from the previous slide
- Then: Get register pair of non-terminal virtmul and generate a Multiply-Accumulate operation **`MADD`** *(☞ chapter 2)*

Complex Example (6)

- C snippet **`a + (b * c)`** with DFT *T* is now *additionally* covered by rules
  dreg: **`tpm_SymbolExp`**
  dreg: **`tpm_BinaryExpPLUS`** (dreg, virtmul)
  virtmul: **`tpm_BinaryExpMULT`** (dreg, dreg)



- Costs for *T*:
  $C[\mathbf{+}] = C[\mathbf{a}] + C[\mathbf{*}] + 1 = C[\mathbf{a}] + (C[\mathbf{b}] + C[\mathbf{c}]) + 1 = 1$

- Code generated for *T*:

```
MADD r_0, r_a, r_b, r_c
```

Invocation of Action Parts with Parameters

`%declare<string>` dreg**`<string target>;`**

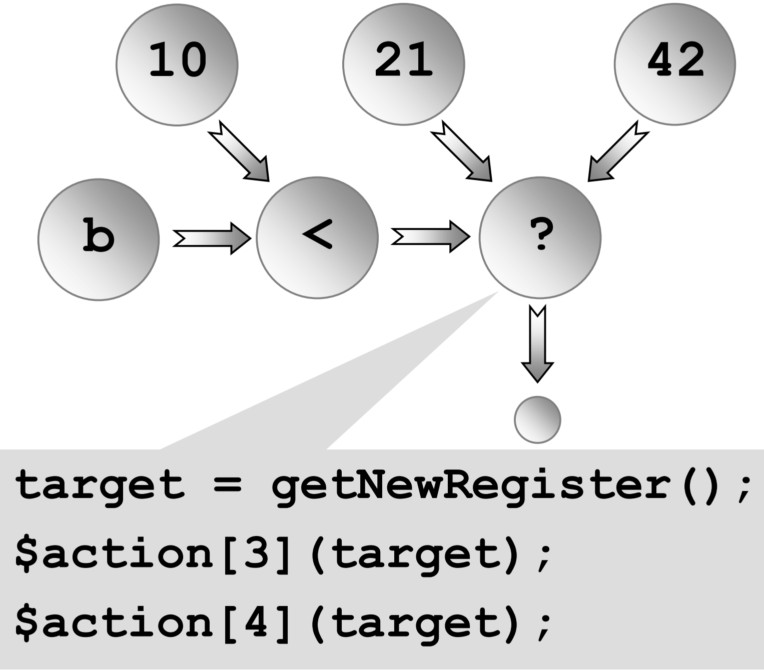
- A **`string`** can be passed as parameter **`target`** to action parts of rules producing a dreg in order to force these action parts to use a specific data register where the target operand shall be stored.

C snippet **`(b < 10) ? 21 : 42`**

- The result of the **`?`** operator must lie in a dreg.

☞ Both sub-trees left and right of the "**`:`**" must be evaluated into the same dreg.

☞ The rule for **`?`** must force both sub-trees to use the very same target register!



```
target = getNewRegister();
$action[3](target);
$action[4](target);
```

Chapter Contents

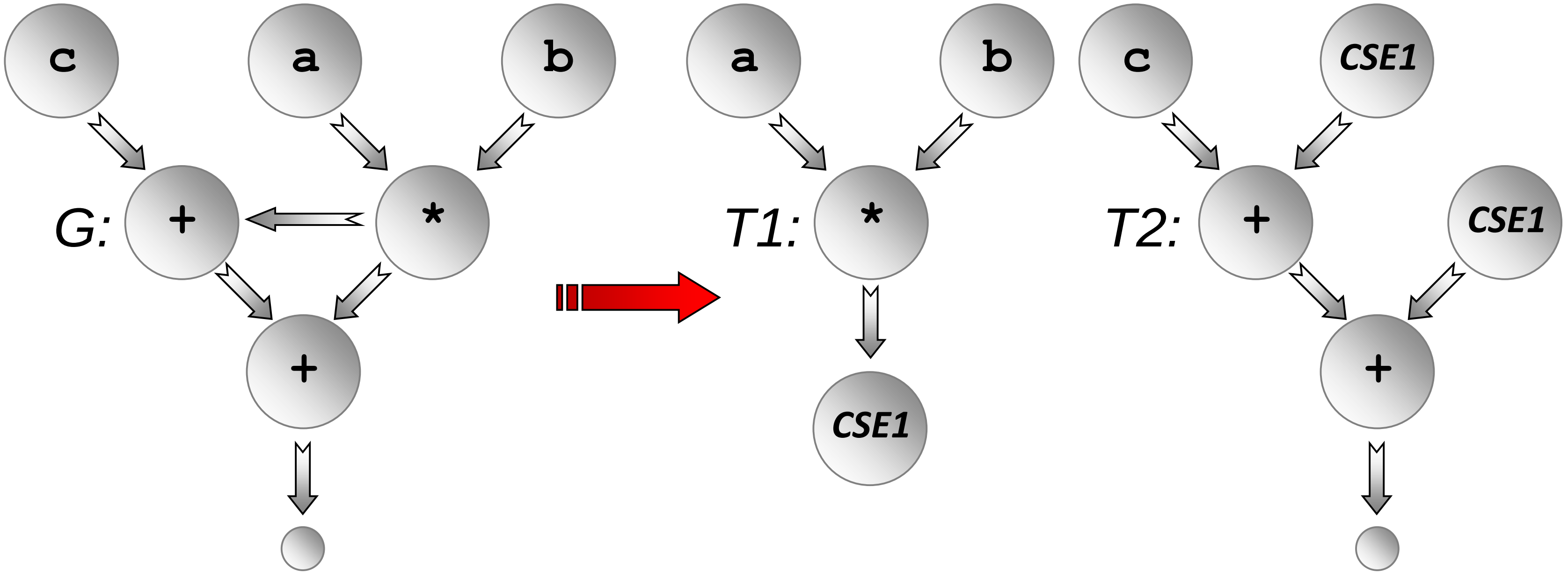
6. Code Generation

- Introduction
  - Role of Code Generation
  - Data Flow Graphs
  - Code Generator Generators
- Tree Covers using Dynamic Programming
  - Partitioning of Data Flow Graphs into Data Flow Trees
  - Tree Covering
  - Tree Pattern Matching Algorithm
  - Tree Grammars for Rule-based Derivation of Code
- Discussion

Limitations of Tree Pattern Matching (1)

Partitioning of DFGs into DFTs yields Sub-Optimal Code

- Example `a + (b * c)` from previous slides is optimally covered by TPM using the `MADD` operation.
- But what happens for, e.g., `e = a * b; ... (c + e) + e ...` ?



Limitations of Tree Pattern Matching (2)

Optimal Tree Cover of *T1* and *T2*

- Would result in a total of *three* machine operations
- 1 multiplication to cover *T1*, 2 additions for *T2*

```
MUL r_0, r_a, r_b
ADD r_1, r_c, r_0
ADD r_2, r_1, r_0
```

Optimal Graph Cover of *G*

- Would result in a total of only *two* machine operations
- 2 multiply-accumulate operations for *G*

```
MADD r_0, r_c, r_a, r_b
MADD r_1, r_0, r_a, r_b
```

Discussion of Tree Pattern Matching

Pros

- Linear run-time complexity
- Optimality for data flow trees
- “Simple” realization using tree grammars and code generator generators

Cons

- TPM only poorly suited for processors with very heterogeneous register files
- TPM inappropriate for processors with parallel processing of instructions

Tree Pattern Matching and Heterogeneous Register Files

Partitioning of DFGs into DFTs

- Let *T* be a DFT that computes a CSE *C*; let *T’* be the DFTs that use *C*.
- After covering *T*, the code generated for *T* must store the value of *C* somewhere, and all *T’* must load this value of *C* from this location.
- Since *T* and *T’* are covered completely independently from each other, the code generation phase for *T* cannot consider where all the *T’* would optimally expect *C*.
- If *T* stores the value of *C* in some part of a heterogeneous register file, but *T’* expects the value of *C* in some different part, additional costly register transfers are necessary!

Tree Pattern Matching and Parallel Processors

Additive Cost Measure of Tree Pattern Matching

- Costs of a DFTs *T* with root *v* are sum of the children's costs plus the costs for *v* itself.
- Action part for *T* usually generates one machine operation.
- *Recall:* Parallel processors execute several machine operations that are grouped into one machine instruction, in parallel.

☞ An additive TPM cost measure that models execution time implicitly assumes that all generated operations are executed purely sequentially!

☞ Since TPM's cost computation is unaware of parallel execution and does not consider that operations can be grouped to instructions, the generated code is likely to have a poor parallel performance!

References

Tree Pattern Matching

- A. Aho, S. Johnson. *Optimal Code Generation for Expression Trees*. Journal of the ACM 23(3), 1976.
- A. Aho, M. Ganapathi, S. Tjiang. *Code Generation Using Tree Matching & Dynamic Programming*. ACM ToPLaS 11(4), 1989.

Code Generator Generators

- *ICD-CG code generator generator*, `http://www.icd.de/en/es/icd-c-compiler/icd-cg`, 2017
- *iburg. A Tree Parser Generator*, `https://github.com/drh/iburg`, 2017. incl. C. W. Fraser, D. R. Hanson, T. A. Proebsting. *Engineering a Simple, Efficient Code Generator Generator*. ACM Letters on Programming Languages and Systems 1(3), 1992.

Summary

Code Generation

- Translation of a DFG into an implementation in the target language
- Code generator generators

Tree Pattern Matching

- Partitioning of into data flow trees
- Linear-time algorithm for optimal DFT covers
- Format and structure of tree grammars

Discussion

- Tree Pattern Matching well-suited only for regular processors
- Disadvantageous for architectures with heterogeneous register files
- Disadvantageous for processors with instruction-level parallelism