Compilation (#6) : Intermediate Representations: CFG, DAGs (Instruction Selection and Scheduling), SSA

Laure Gonnord & Matthieu Moy & other

`https://compil-lyon.gitlabpages.inria.fr/compil-lyon/`

Master 1, ENS de Lyon et Dpt Info, Lyon1

2020-2021

Big picture

$$\begin{tikzcd}
\text{source code} \arrow[d] \\
\boxed{\text{lexical+syntactic analysis + typing}} \arrow[d] \\
\text{decorated AST} \arrow[d] \\
\boxed{\text{code production (numerous phases)}} \arrow[d] \\
\text{assembly language}
\end{tikzcd}$$

In context 1/2

In the last course we saw the need for a better data structure to propagate and infer information. We need:

- A data structure that helps us to reason about the flow of the program.
- Which embeds our three address code.

▶ Control-Flow Graph.

In context 2/2

$$\begin{tikzcd}
\text{decorated AST} \arrow[d] \\
\boxed{\text{IR Construction}} \arrow[d] \\
\text{Control-Flow Graph} \arrow[d] \\
\boxed{\text{Clever analyses/code generation}} \arrow[d] \\
\text{assembly language}
\end{tikzcd}$$

Definitions

Definition (Basic Block)

Basic block: largest (3-address RISCV*) instruction sequence without label. (except at the first instruction) and without jumps and calls.*

Definition (CFG)

It is a directed graph whose vertices are basic blocks, and edge $B_1 \rightarrow B_2$ exists if B_2 can follow immediately B_1 in an execution.

- two optimisation levels: local (BB) and global (CFG)

An example 1/2

Let us consider the program:

```
int x,y;
if (x<4) y=7; else y=42;
x=10;
```

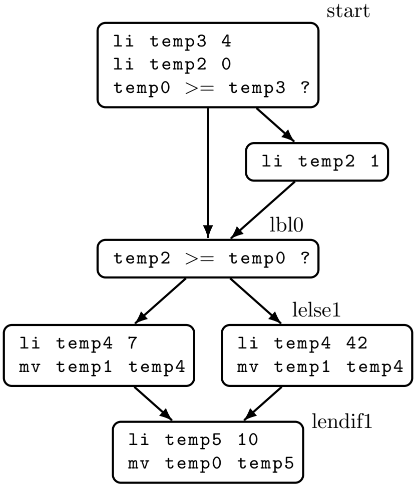
We already generated the (linear code) for a large part of it.

An example 2/2

```
  li  temp3, 4
  li  temp2, 0
  geq temp0, temp3, lbl0
  li  temp2, 1
lbl0 :   # if  false ,  jump
  eq tmp2, zero, lelse1
  li  temp4, 7
  mv temp1, temp4  # y gets 7
  jump lendif1
lelse1 :
  li  temp4 42
  mv temp1, temp4  # y gets 42
lendif1 :
  li  temp5, 10
  mv temp0, temp5  # end
```

An example 2/2

```
  li  temp3, 4
  li  temp2, 0
  geq temp0, temp3, lbl0
  li  temp2, 1
lbl0 :   # if  false , jump
  eq tmp2, zero, lelse1
  li  temp4, 7
  mv temp1, temp4  # y gets 7
  jump lendif1
lelse1 :
  li  temp4 42
  mv temp1, temp4  # y gets 42
lendif1 :
  li  temp5, 10
  mv temp0, temp5  # end
```



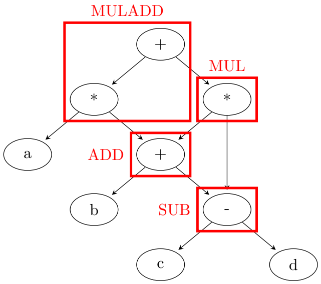
Identifying Basic Blocks (from 3 address code)

- The first instruction of a basic block is called a **leader**.
- We can identify leaders via these three properties:
  1. The first instruction in the intermediate code is a leader.
  2. Any instruction that is the target of a conditional or unconditional jump is a leader.
  3. Any instruction that immediately follows a conditional or unconditional jump is a leader.
- Once we have found the leaders, it is straighforward to find the basic blocks: for each leader, its basic block consists of the leader itself, plus all the instructions until the next leader.

Big picture (Basic Block Optimisation)

- Front-end → a CFG where nodes are basic blocks.
- Basic blocks → DAGs that explicit common computations

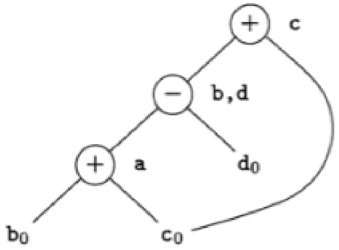
```
u1  := c - d
u2  := b + u1
u3  := a * u2
u4  := u2 * u1
u5  := u3 + u4
```



▶ choose instructions(**selection**) and order them (**scheduling**).

An Example of BB DAG construction

```
a = b + c
b = a - d
c = b + c
d = a - d
```



Useful links :
`https://www.youtube.com/watch?v=PXTKWvyQUwE` and `https://www.cse.iitm.ac.in/~krishna/cs3300/pm-lecture3.pdf`
for other BB optimisations.

Instruction Selection, in general

The problem:

- a list of instructions/operations that compute one or more expressions.
- map these operations in “real machine instructions”.
- at minimum cost.

Instruction Selection

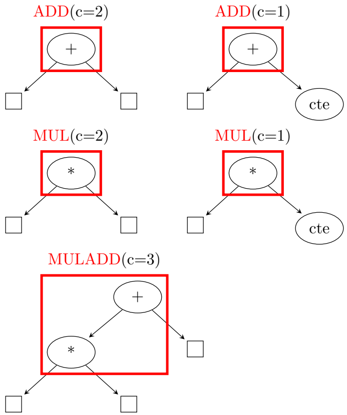
The problem of selecting instructions is a DAG-partitioning problem. But what is the objective ?

The best instructions:

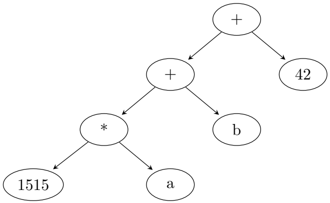
- cover bigger parts of computation.
- cause few memory accesses.

► Assign a cost to each instruction, depending on their addressing mode.

Instruction Selection: an example



What is the optimal instruction selection for:



- Finding a tiling of minimal cost: it is **NP-complete** (SAT reduction).

Tiling trees / DAGs, in practice

For tiling:

- There is an optimal algorithm for **trees** based on dynamic programming.
- For DAGs we use heuristics (decomposition into a forest of trees, . . .)

▶ The literature is plethoric on the subject.

Instruction Selection, in our compiler

Mapping one to one. No real choice.

Instruction Scheduling, in general

The problem:

- change the order of instructions.
- to “optimise’.
- without “cutting dependencies”.

Instruction Scheduling, what for?

We want an evaluation order for the instructions that we choose with **Instruction Scheduling**.

A scheduling is a function θ that associates a **logical date** to each instruction. To be correct, it must respect data dependancies:

```
(S1) u1 := c - d
(S2) u2 := b + u1
```

implies $\theta(S_1) < \theta(S_2)$.

▶ How to choose among many correct schedulings? depends on the target architecture.

Architecture-dependant choices

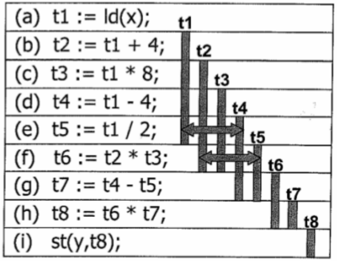
The idea is to exploit the different ressources of the machine at their best:

- instruction parallelism: some machine have parallel units (subinstructions of a given instruction).
- prefetch: some machines have non-blocking load/stores, we can run some instructions between a load and its use (hide latency!)
- pipeline.
- registers: see next slide.

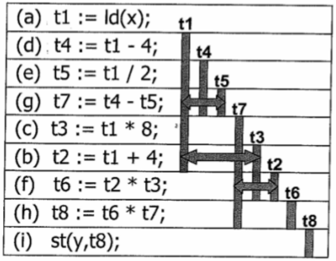
(sometimes these criteria are incompatible)

Register use

Some schedules induce less **register pressure**:



```
(a) t1 := ld(x);
(b) t2 := t1 + 4;
(c) t3 := t1 * 8;
(d) t4 := t1 - 4;
(e) t5 := t1 / 2;
(f) t6 := t2 * t3;
(g) t7 := t4 - t5;
(h) t8 := t6 * t7;
(i) st(y,t8);
```



```
(a) t1 := ld(x);
(d) t4 := t1 - 4;
(e) t5 := t1 / 2;
(g) t7 := t4 - t5;
(c) t3 := t1 * 8;
(b) t2 := t1 + 4;
(f) t6 := t2 * t3;
(h) t8 := t6 * t7;
(i) st(y,t8);
```

- How to find a schedule with less register pressure?

Scheduling wrt register pressure ENSL Only

Result: this is a linear problem on trees, but NP-complete on DAGs (Sethi, 1975).

- Sethi-Ullman algorithm on trees, heuristics on DAGs

Sethi-Ullman algorithm on trees ENSL Only

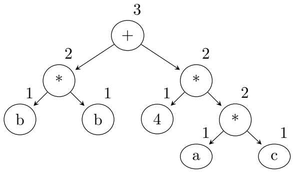
$\rho(node)$ denoting the number of (pseudo)-registers necessary to compute a node:

- $\rho(leaf) = 1$
- $\rho(nodeop(e_1, e_2)) = \begin{cases} max\{\rho(e_1), \rho(e_2)\} & \text{if } \rho(e_1) \neq \rho(e_2) \\ \rho(e_1) + 1 & \text{else} \end{cases}$

(the idea for non “balanced” subtrees is to execute the one with the biggest ρ first, then the other branch, then the op. If the tree is balanced, then we need an extra register)

▶ then the code is produced with postfix tree traversal, the biggest register consumers first.

Sethi-Ullman algorithm on trees - an example



	tmp_1	tmp_2	tmp_3	tmp_4
`mul tmp1, b, b`				
`mul tmp2, a, c`	■			
`leti tmp3, 4`	■	■		
`mul tmp4, tmp2, tmp3`	■	■	■	
`add tmp5, tmp1, temp4`	■			■

Instruction Selection, in our compiler

Same order as the 3-address code.

Conclusion (instruction selection/scheduling)

Plenty of other algorithms in the literature:

- Scheduling DAGs with heuristics, . . .
- Scheduling loops (M2IF course on advanced compilation)

Practical session:

- we have (nearly) no choice for the instructions in the RISCV ISA.
- evaluating the impact of scheduling is a bit hard.

We won't implement any of the previous algorithms.

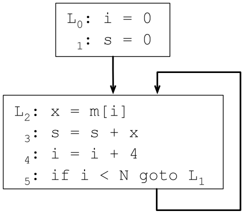
TODO Yannick

Credits

Source `http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/StaticSingleAssignment.pdf`

The Static Single Assignment Form

- This name comes out of the fact that each variable has only one definition site in the program.
- In other words, the entire program contains only one point where the variable is assigned a value.
- Were we talking about Single Dynamic Assignment, then we would be saying that during the execution of the program, the variable is assigned only once.



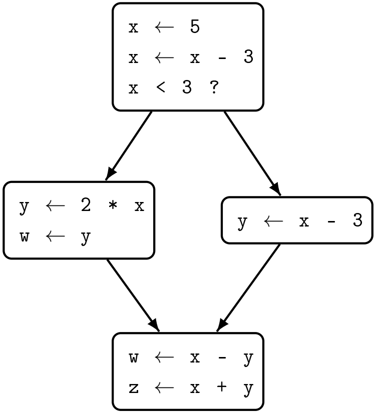
```
L0: i = 0
 1: s = 0
```

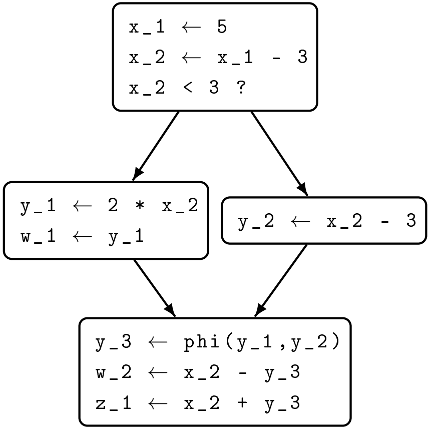
```
L2: x = m[i]
 3: s = s + x
 4: i = i + 4
 5: if i < N goto L1
```

Variable i has two static assignment sites: at L_0 and at L_4; thus, this program is not in Static Single Assignment form. Variable s, also has two static definition sites. Variable x, on the other hand, has only one static definition site, at L_2. Nevertheless, x may be assigned many times dynamically, i.e., during the execution of the program.

A first Example (Cytron 1991)

Each variable is assigned only once (Static Single Assigment form):





Pro/cons

- - Another IR, and cost of contruction/deconstruction
- + (some) Analyses/optimisations are easier to perform (like register allocation): `http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/SSABasedRA.pdf`

Summary