Instruction Scheduling

Mooly Sagiv
Tel Aviv University
and
Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

July 14, 1997

Instruction Scheduling

- Reorders instruction stream as generated by instruction selection,
- Goal: Exploitation of intraprocessor parallelism,
  - Filling very long instruction words (VLIWs), or
  - Avoiding resource collisions and pipeline hazards.
- Must be semantics preserving,
- Basis: Program dependences.

Program Dependences

- Relation between definitions and uses of resources (program variables, memory cells or blocks, symbolic or real registers),
- Here mainly machine resources, i.e. registers, memory cells, status words.
- Alias problems:
  - Address calculation for an index expression,
  - Dereferencing of a pointer.

Definitions and Uses of Machine Resources

Definitions :

- modifications of register contents by loads or operations, pre-, postincrement/decrement,
- setting carry, overflow, condition bits in status words,
- storing values in memory cells,
- modifying registers as side effects of e.g. pop, push.

Uses :

- Using register contents in operations and for addressing,
- Storing register contents,
- Loading contents of memory cells,
- Testing the program status word.

Types of Dependences

A sequence of definitions ($X :=$) and uses ($:= X$) of X.

$$\begin{array}{lll} a: & X & := \\ b: & X & := \\ c: & & := X \\ d: & X & := \end{array}$$

Output dependences (dd): Definitions on definitions, e.g., b on a,

True dependence (du): Uses on definitions, e.g., c on b,

Antidependence (ud): Definitions on uses, e.g., d on c.

Program Dependence Graph (PDG)

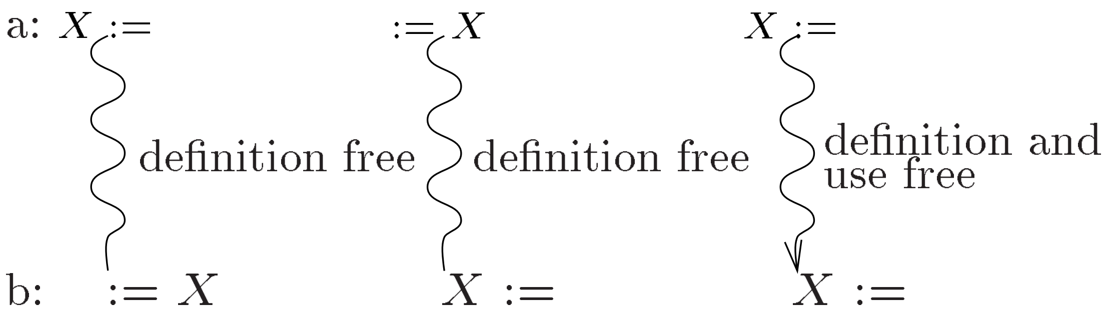
For a basic block.

Nodes instructions,

Edges

- a sets a resource, b uses it, and the path from a to b is definition free, or
- a uses a resource, b sets it, and the path from a to b is definition free, or
- a and b set the same resource and the path from a to b is use and definition free

describes the maximal degree of freedom for semantics preserving reordering of the instructions.

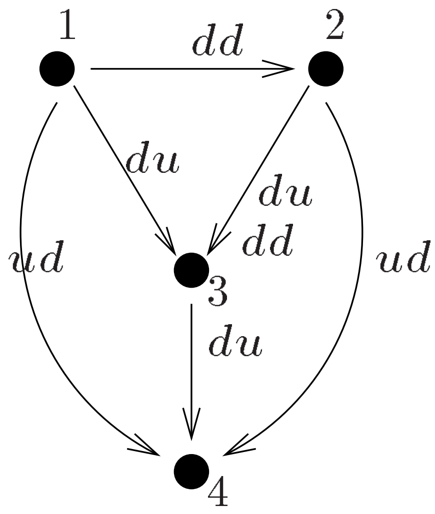


Example

PDG

- does not contain all dependencies as edges,
- but transitively,
- enough to prevent non–semantic preserving reorderings.

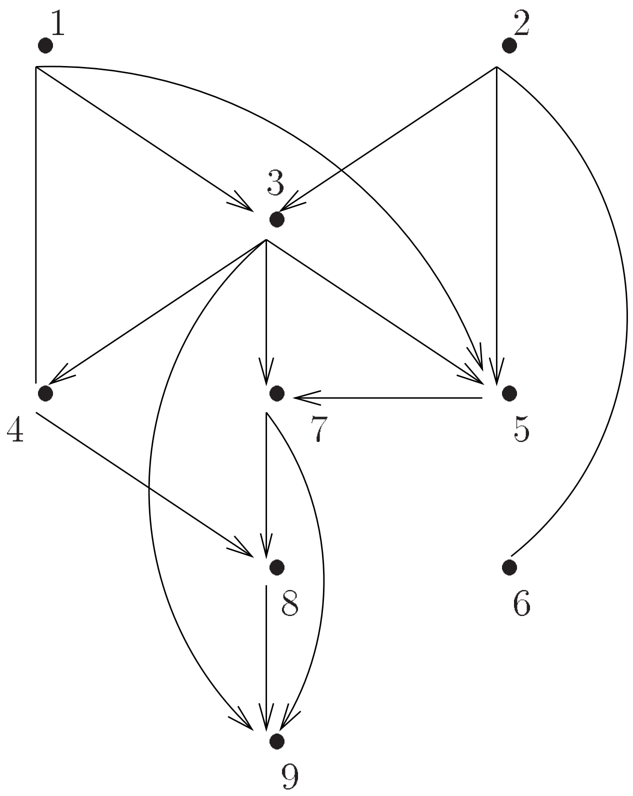
Instruction sequence with its PDG



$1:$	(CC, D1)	:= M[A1 + 4].W
$2:$	(CC, D2)	:= M[A1 + 6].W
$3:$	(CC, D1)	:= D1 + D2
$4:$	M[A1 + 4]	:= D1.W

Basic Block with PDG

1:	D1	:= M[A1+4];
2:	D2	:= M[A1+6];
3:	A1	:= A1+2;
4:	D1	:= D1+A1;
5:	M[A1]	:= A1;
6:	D2	:= D2+1;
7:	D3	:= M[A1+12];
8:	D3	:= D3+D1;
9:	M[A1+6]	:= D3



Algorithm PDG-Graph

Input: basic block
Output: dependence graph of basic block
Method:
var $lastDefs$, $expUses$**: set of pair** ($resource$, $instrOcc$)**;**
 $actInstr$: $instruction$;

function $conflict(resource, instr_1, instr_2) : conflictTyp$;
 determines existence and type of conflict between $instr_1$ and $instr_2$ on $resource$.

procedure $drawEdge(a \rightarrow b)$
 draws a new edge between its arguments if there is none.

begin
 $actInstr :=$ last instruction of basic block;
 $createNode(actInstr)$;
 $lastDefs := \{(r, actInstr) | r \in defs(actInstr)\}$;
 $expUses := \{(r, actInstr) | r \in uses(actInstr)\}$;
 while $pred(actInstr)$ $defined$ **do**
  $actInstr := pred(actInstr)$;

```
foreach resource r set or used in actInstr do
  foreach (r, b) ∈ lastDefs ∪ expUses do
    case conflict(r, actInstr, b) of
    def-def: if exists no pair (r, α) in expUses
      then drawEdge(actInstr → b);
      fi;
    def-use: drawEdge(actInstr → b);
    use-def: drawEdge(actInstr → b);
    end case;
  od
od;
foreach resource r' set in actInstr do
  lastDefs := lastDefs −
    {s ∈ lastDefs | ∃α : s = (r', α)} ∪ {(r', actInstr)};
  expUses := expUses −
    {b ∈ expUses | ∃β : b = (r', β)};
od;
foreach resource r' used in actInstr do
  expUses := expUses ∪ {(r', actInstr)};
od;
od
end
```

Pipeline Scheduling

- instruction pipeline without hazard detection,
- one cycle delay (simplification) for dependent instructions,
- goal: minimization of pogram length,
- NP-complete problem,
- use topological sorting with heuristics for the selection of candidates next to be scheduled:
  - candidates with most dependences,
  - candidates on the longest path.

Algorithm Pipeline Scheduling

Input: Basic block with PDG,
set of schedules for preceding basic blocks.
Output: (Possibly) reordered instruction sequence of the basic block, possibly with inserted NOPs.
Method: topol. sorting constrained by the pipeline conds.

var $cands$, $realCands$, $potColls$:
set of $instrOcc$;

function $colliding(cand, potCol)$: **set of** $instrOcc$;
computes the set of instructions in $cand$,
colliding with those in $potCol$

begin
$cands$:= set of min. El. of the PDG;
$potColls$:= set of last instructions in schedules of
predecing basic blocks;
repeat
$realCands := cands - colliding(cands, potColls)$;
if $realCands \neq \emptyset$ **then**
evaluate candidates according to heuristics;
select a best candidate b; schedule b;
remove b from $cands$;

remove b and all outgoing edges from the PDG;
insert new nodes without predecessors into $cands$;
$potColls := \{b\}$
else schedule a NOP; $potColls := \emptyset$
fi
until $cands = \emptyset$
end