Code Generation: Integrated Methods

- Integration of register allocation with instruction selection,

- Simple machine model:
  - r general purpose register $R_0, \ldots, R_{r-1}$,
  - Two address instructions

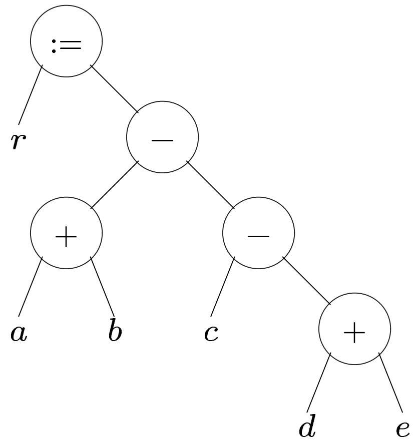
R_i	$:=$	$M[V]$	Load
$M[V]$	$:=$	R_i	Store
R_i	$:=$	$R_i \; op \; M[V]$	Compute
R_i	$:=$	$R_i \; op \; R_j$	

- Two phases:
  1. Computing register requirements,
  2. Generating code, allocating registers and temporaries.

Example Tree

Source $r := (a + b) - (c - (d + e))$

Tree



Generated Code

2 Registers R_0 and R_1
Two possible code sequences:

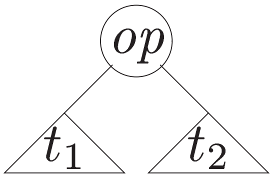
R_0	$:=$	$M[a]$	R_0	$:=$	$M[c]$
R_0	$:=$	$R_0 + M[b]$	R_1	$:=$	$M[d]$
R_1	$:=$	$M[d]$	R_1	$:=$	$R_1 + M[e]$
R_1	$:=$	$R_1 + M[e]$	R_0	$:=$	$R_0 - R_1$
$M[t_1]$	$:=$	R_1	R_1	$:=$	$M[a]$
R_1	$:=$	$M[c]$	R_1	$:=$	$R_1 + M[b]$
R_1	$:=$	$R_1 - M[t_1]$	R_1	$:=$	$R_1 - R_0$
R_0	$:=$	$R_0 - R_1$	$M[f]$	$:=$	R_1
$M[f]$	$:=$	R_0			

- Left code
  - stores result for $c - (d + 2)$ in a temporary,
  - no register available,

- Right code
  - evaluates $c - (d + 2)$ first (needs 2 registers),
  - saves one instruction,

The Algorithm

Principle: Given tree t for expression $e_1 \ op \ e_2$



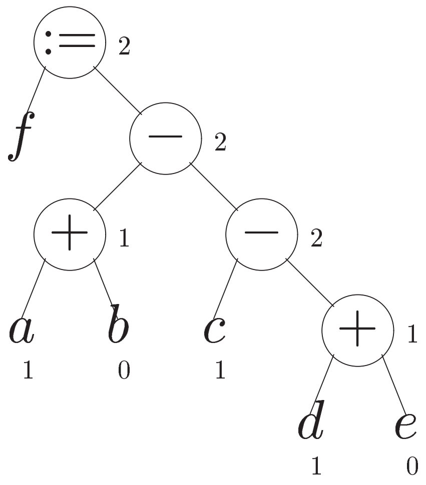
- t_1 needs r_1 registers, t_2 needs r_2 registers,
- Let $r \geq r_1 > r_2$:
  After evaluation of t_1:
  - $r_1 - 1$ registers freed,
  - one holds the result,

  t_2 gets enough registers to evaluate, hence
  t can be evaluated in r_1 registers,
- Let $r_1 = r_2$:
  t needs $r_1 + 1$ registers to evaluate,
- Let one of t_1 or t_2 need more than r registers,
  spill to temporary required.

Labeling Phase

- Labels each node with its register needs,
- Bottom up pass,
- Left leaves labeled with '1', have to be loaded into register,
- Right leaves labeled with '0', are used as operands,
- Inner nodes:
  $$regneed(op(t_1, t_2)) = \begin{cases} max(r_1, r_2), & \text{if } r_1 \neq r_2 \\ r_1 + 1, & \text{if } r_1 = r_2 \end{cases}$$
  where $r_1 = regneed(t_1)$, $r_2 = regneed(t_2)$

Example



Generation Phase

Principle:

- Generates instruction **Op** for operator op in $op(t_1, t_2)$ after generating code for t_1 and t_2.
- Order of t_1 and t_2 depends on their register needs,
- Upon execution of the generated **Op**–instruction: value of t_1 in register,
- $RSTACK$ – available registers, init. all registers,
  Before processing t: *top*($RSTACK$) is result register for t,
  After processing t: all registers available,
  top($RSTACK$) is result register for t.
- $TSTACK$ – available temporaries.

Algorithm Gen_Opt_Code

Algorithm	$RSTACK$-Contents resp. result register
var $RSTACK$: **stack of register**;	
var $TSTACK$: **stack of address**;	
proc Gen_Code $(t:$ **tree**$)$;	$(R', R'', \ldots)$
var R: **register**, T: **address**;	
case t **of**	
$(leaf\ a, 1)$: (∗left leaf∗)	
$emit(top(RSTACK)\ := a)$;	result in R'
$op((t_1, r_1), (leaf\ a, 0))$: (∗right leaf∗)	
$Gen\_Code(t_1)$;	
$emit(top(RSTACK)\ :=\ top(RSTACK)$ **Op** $a)$;	result in R'

$op((t_1, r_1), (t_2, r_2))$:	$(R', R'', \ldots)$
cases	
$r_1 < \ min(r_2, r)$:	$(R', R'', \ldots)$
begin	
exchange$(RSTACK)$;	$(R'', R', \ldots)$
$Gen\_Code(t_2)$;	result in R''
$R := $ **pop**$(RSTACK)$;	$(R', \ldots)$
$Gen\_Code(t_1)$;	result in R'
$emit(top(RSTACK) \ := \ top(RSTACK)$ **Op** $R)$;	result in R'
push$(RSTACK, R)$;	$(R'', R', \ldots)$
exchange$(RSTACK)$;	$(R', R'', \ldots)$
end ;	

$r_1 \geq r_2 \wedge r_2 < r$:	$(R', R'', \ldots)$
begin	
$Gen\_Code(t_1)$;	result in R'
$R :=$ **pop**$(RSTACK)$;	$(R'', \ldots)$
$Gen\_Code(t_2)$;	result in R''
$emit(R\ :=\ R$ **Op** $top(RSTACK))$;	result in R'
push$(RSTACK, R)$;	$(R', R'', \ldots)$
end ;	

$r_1 \geq r \wedge r_2 \geq r$:	$(R', R'', \ldots)$
begin	
$Gen\_Code(t_2)$;	result in R'
$T :=$ **pop**$(TSTACK)$;	
$emit(M[T] := top(RSTACK))$;	result in $M[T]$
$Gen\_Code(t_1)$;	result in R'
$emit(top(RSTACK) := top(RSTACK)$ **Op** $M[T])$;	result in R'
push$(TSTACK, T)$;	
end ;	
endcases	
endcase	
endproc	

Register Allocation and Instruction Selection by Dynamic Programming

- More complex architecture,
  - r general purpose registers $R_0, \ldots, R_{r-1}$,
  - Instruction formats:

R_i	$:=$	e	Compute
R_i	$:=$	$M[V]$	Load
$M[V]$	$:=$	R_i	Store

    e term with registers and memory cells,
    costs associated with each instruction.

- Goal: Generate cheapest instruction sequence using no more than r registers.

- Assume contiguous computation of subtrees $\Longrightarrow$ only one register required to hold the result

- Use instruction selection techniques with tree parsing,
  compute cheapest derivation.

Canonical recursive solution

- Assume e of instruction $R_i := e$ matches tree t
- subtrees of t corresp. to memory operands of e: computed into memory, no registers occupied after that
- let e have k register operands
- how to compute the corresponding subtrees $t_1, t_2, \ldots, t_k$ into these registers?
- assume order $i_1, i_2, \ldots, i_k$ and j available registers
- t_{i_1} has j registers available, t_{i_2} has $j-1$, t_k has $j-k$
- if this fits $(j - k - regneed(t_{i_k}) \geq 0)$, add the minimal costs for computing all subtrees in this way to the costs of e to yield the minimal costs for this combination

- if not enough registers are available, compute enough subtrees into memory, sum up costs like above

Doing it for all potential combinations recomputes the costs for subtrees $\Longrightarrow$ exponential complexity

Dynamic Programming

- Convert top-down algorithm into bottom-up algorithm tabulating partial solutions

- Associate cost vector $C[0..r]$ with each node n,
  C[0] cheapest costs for computing t/n into a temporary,
  C[i] cheapest costs computing t/n into a register using i registers.

- Compute cost vector at node n minimizing over all “legal” combinations of

  – one applicable instruction,
  – the cost vectors of the nodes “under” non–terminal nodes in the applied rule.

- What is a “legal” combination for $C[j], j > 0$?
  Any combination of generated code for subtrees not needing more than j registers.

- Extract cheapest instruction sequence in a second pass.

Global Register Allocation

- Register allocation across whole procedures/programs,
- **Symbolic Registers** for intermediate results and modified variables,
- Two symbolic registers **collide** if their contents are "live" at the same time,
- Colliding symbolic registers cannot be allocated to the same real register,
- Goal: allocate the (unbounded number of) symbolic registers to the fixed number of real registers without collision.

Definitions

- A **definition** of a symbolic register is the computation of an intermediate result or the modification of a variable,
- A **use** of a symbolic register is a reading access to the corresponding variable or a use of the intermediate value,
- A **definition–path** of r to program point p is a path from the starting point to p with a definition of r,
- A **use–path** from p is a definition free path starting at p containing a use of r,
- Symbolic register r is **live** at program point p if exists a definition–path to p and a use–path from p,
- The **life span** of r is the set of all program points, on which r is live.
  Value of a live symbolic register may still be used.

- Two life spans of symbolic registers **collide** if one of the registers is set in the life span of the other.

- **Register Collision Graph**
  - nodes – life spans,
  - edge between colliding life spans.

- Allows viewing register allocation as graph colouring.

Graph Colouring

- k real registers available,
- Solve k–colouring problem,
- NP–complete for $k > 2$,
- Use heuristics.

Algorithm:
Case:
G contains a node n with degree $< k$.
n and its neighbours can be coloured with different colours.
Remove n from G, decreasing the size of G.
G is k-colourable if we arrive at the empty graph.

Case:
G not empty and exists no node with degree $< k$:
use heuristics to select one node to remove (spilling),
modify program inserting spills at definitions and loads at uses,
reflect change in graph.

Heuristics for Node Removal

1. degree of the node: high degree causes many deletions of edges,

2. costs of spilling.

Example

Input-program

1; $x := 1;$
2: $y := 2;$
3: $z := x + y;$
4: $x := x + z;$
5: print x

Register collision graph

Instruction Level Parallelism (ILP)

Simultaneous execution of multiple instructions

- issued at the same time as statically scheduled by a compiler: VLIW
- issued while preceding instructions still execute: pipelining
- combination possible: VLIW architecture with pipelined functional units

The VLIW Architecture

- Several functional units,
- One instruction stream,
- Jump priority rule,
- FUs connected to register banks,
- Enough parallelism available?

Instruction Pipeline

Several instructions in different states of execution
Potential structure:

1. instruction fetch and decode,

2. operand fetch,

3. instruction execution,

4. write back of the result into target register.

		cycle						
		1	2	3	4	5	6	7
Pipe-	1	B_1	B_2	B_3	B_4			
line-	2		B_1	B_2	B_3	B_4		
stage	3			B_1	B_2	B_3	B_4	
	4				B_1	B_2	B_3	B_4

Pipeline hazards

- Data hazards: Needed operand not available,
- Structural hazards: Resource conflicts,
- Control hazards: (Conditional) jumps.

Phases in dynamically scheduled execution

1. fetch
2. decode
   instruction

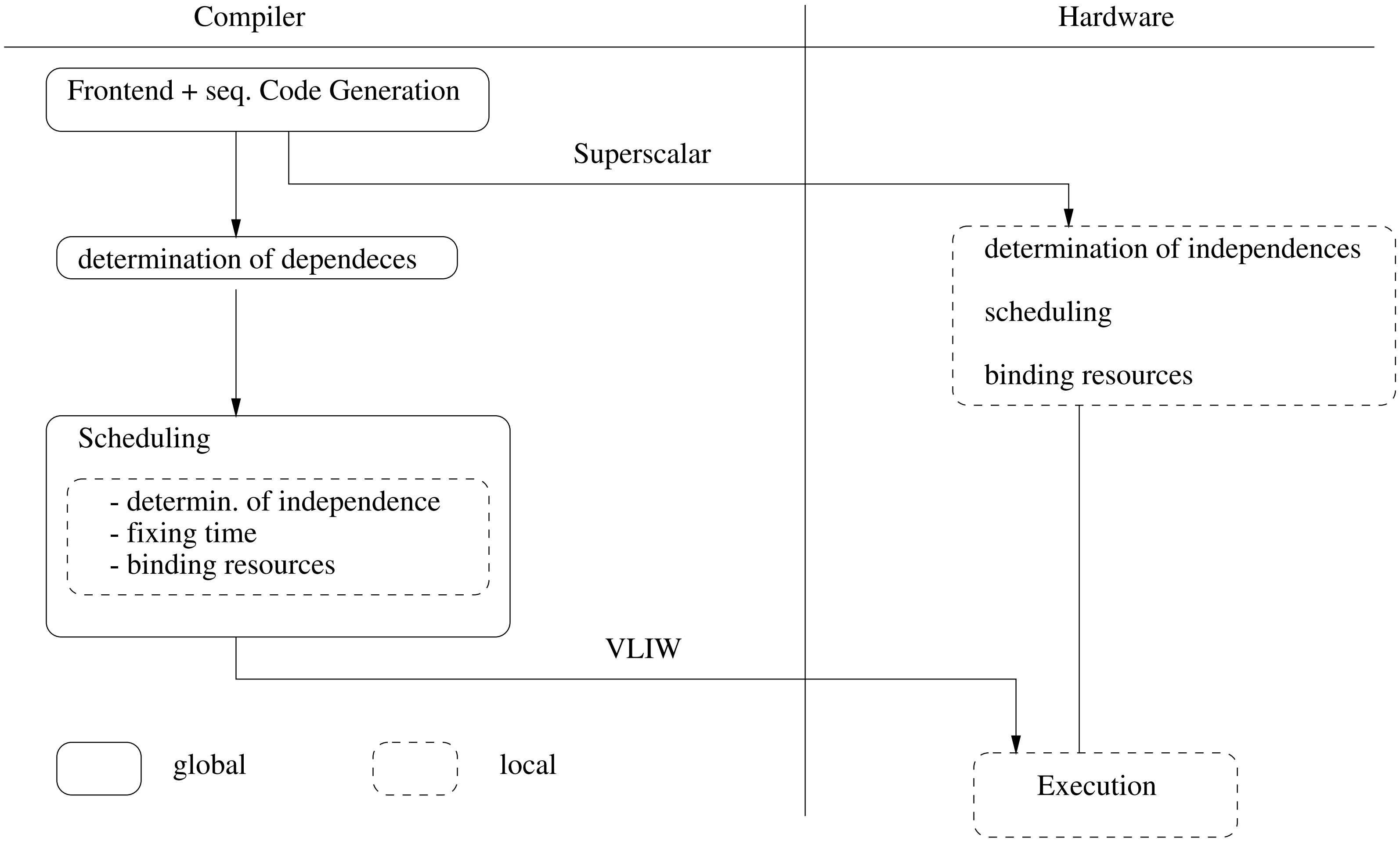
   detection of structural hazards

   operand

   detection of data hazards
3. operand fetch

   dispatch (issue)
4. execute

Architecture Characterization



Static and Dynamic Scheduling

Static	Dynamic
1. global dependence analysis.	in each scheduling step:
2. in each scheduling step:	with local dependence analysis,
check non-dep. of candidates	check non-dep. of candidates
on prev. scheduled instructions.	on curr. executing or delayed instructions.
schedule non-dep. instructions	dispatch or delay non-dep. instructions.
after appropriate delay	

Instruction Scheduling

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

List Scheduling (1)

SET $data\_ready$;
INT $cycle = 0$;

Insert operations without predecessors in the data dependence graph into the set $data\_ready$.

while $(data\_ready \neq \emptyset)$ **do**
  cycle $=$ cycle $+ 1$;

  Choose operations from $data\_ready$ in priority order and insert them in the current cycle, until $data\_ready$ is empty or the insertion leads to a resource conflict.

  Insert all operations into $data\_ready$ that can be scheduled in the next cycle without violating data dependences.

od

List Scheduling (2)

- The priority in which operations from the data ready set are chosen is determined by heuristics.
- Common heuristics: *Highest-Level First*.
  The priority of each operation is the length of the longest path in the data dependence graph starting from this operation.

Exposing more Instruction Level Parallelism

Degree of ILP in basic blocks is limited – typically to 2

Available ILP in processors grows: better exploitation by

- Scheduling sequences of consecutive basic blocks
- Scheduling loops
- Speculation

Speculation

dynamic: based on hardware branch prediction, mispredicted branch – forgetting or undoing effects of speculatively executed instructions

static: generating compensation code for "speculatively placed" instructions

Trace/Superblock Scheduling

- Profile-directed optimization and scheduling of sequences of consecutive basic blocks (*traces, superblocks*)
- Program annotated with profiling information: Each branch of a conditional associated with a relative frequency
- Used to identify frequently taken paths
- Frequently taken traces optimized and scheduled at the cost of less frequently taken traces

Trace Scheduling

- A *trace* is a sequence of consecutive basic blocks not extending across loop boundaries.

- The control flow graph of a procedure is partitioned into a disjoint set of traces.

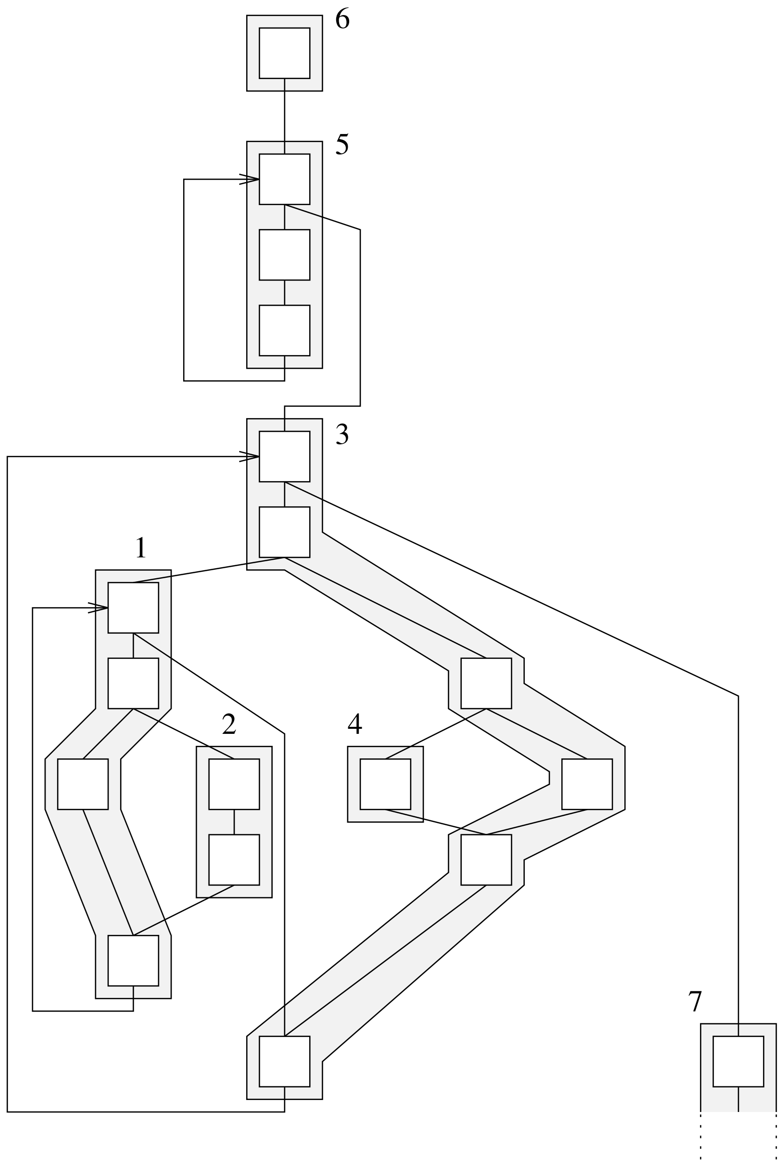
- The traces are formed in order of decreasing frequency:

  1. Select available basic block with highest frequency
  2. Join available predecessors and successors with highest frequencies until frequency falls below a given threshold

- There are

  side exits out of traces
  side entrances into traces

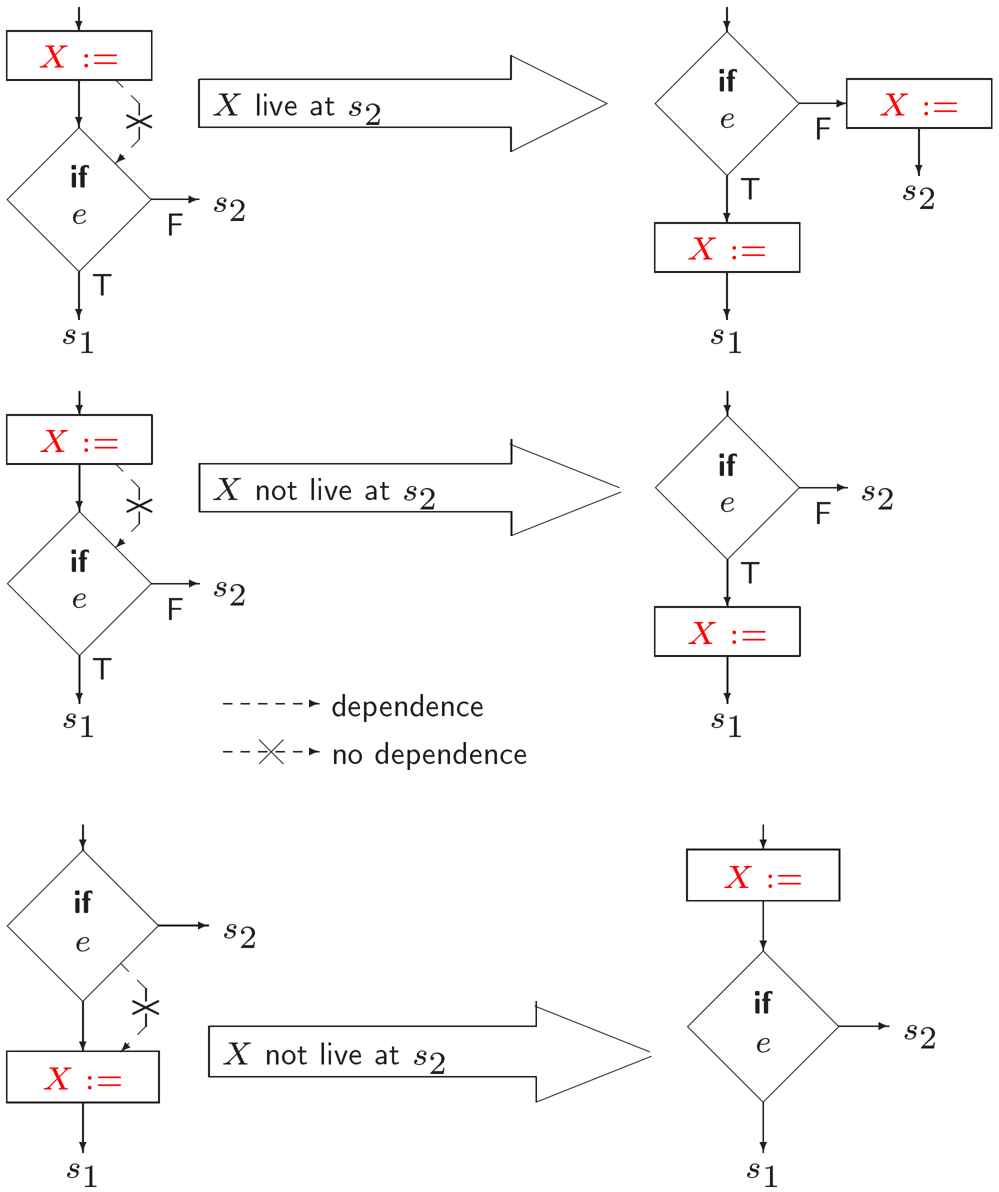
A Partitioning into Traces



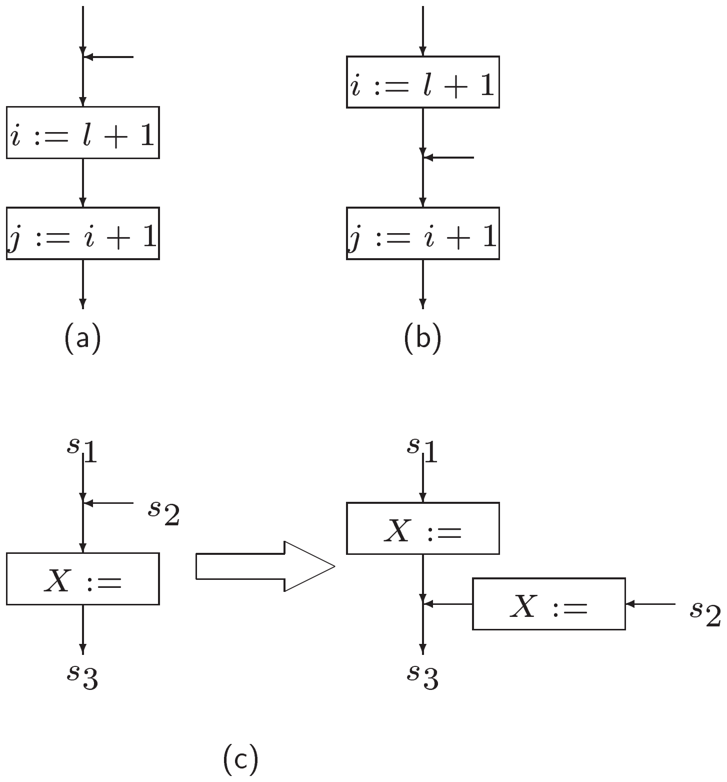
Trace Scheduling

- List scheduling of a trace leads to problems: the semantics may be destroyed by
  - code motion past side exits
  - code motion past side entrances.
- Consequence: *Compensation code* inserted on off-trace paths.
- Problems of compensation code:
  - Code growth
  - Exceptions raised by compensation code moved in front of side exits

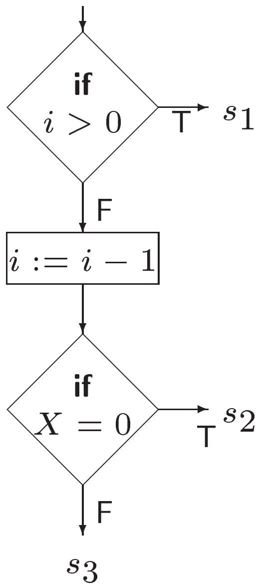
Moving Statements across Conditional Branches



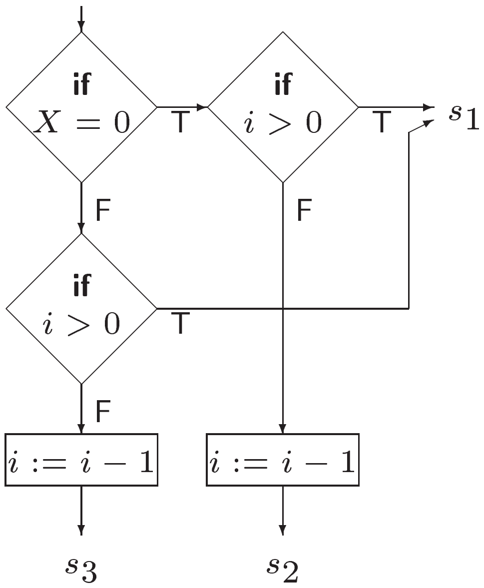
Moving a Statment past a Side Entrance



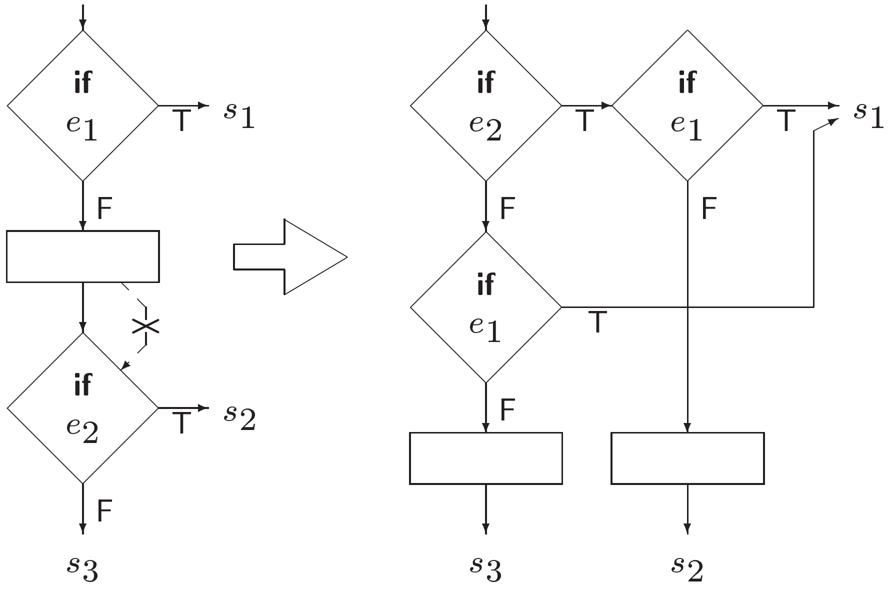
Exchanging Conditional Branches

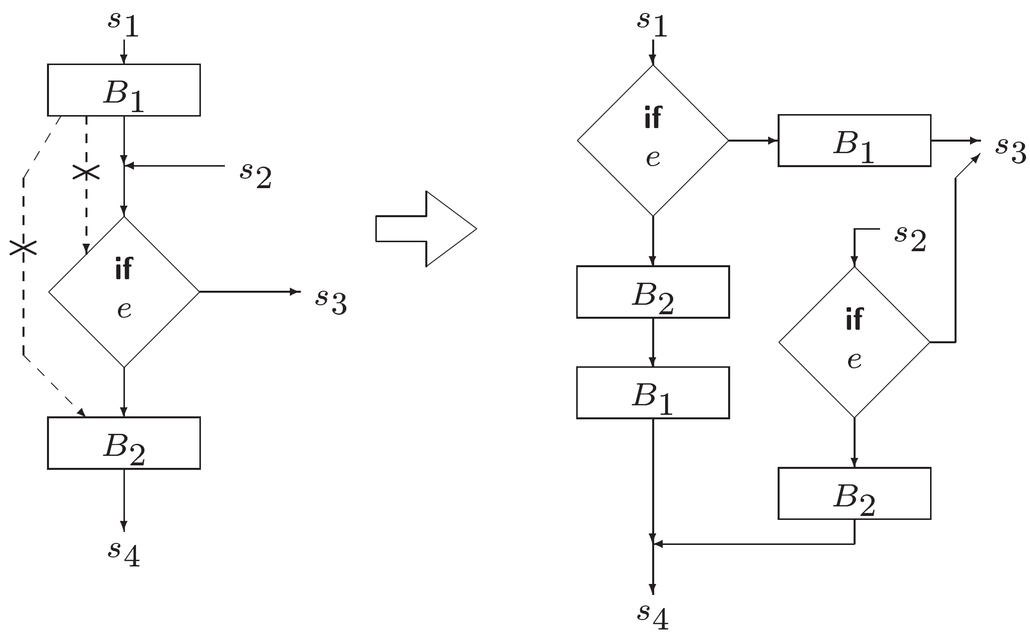


(a)



(b)


if
e1
T
s1
F
if
e2
T
s2
F
s3
if
e2
T
if
e1
T
s1
F
F
if
e1
T
F
s3
s2


s1
B1
s2
if
e
s3
B2
s4
s1
if
e
B1
s3
B2
B1
s2
if
e
B2
s4

Superblock Scheduling

- Goal: Avoiding code motion past side entrances by *tail duplication*:
  copying code starting with side entrance and redirecting the branches.

- Superblock formation:

  - starts with a trace
  - produces a trace without side entrances

- Compensation code only for code motion past side exits.