Functional units, delay slots, and dependency analysis

Architecture-Dependent Optimization
more cycles have elapsed.

Execution cannot start at destination before one or more instructions have arrived at the instruction cache. If an instruction awaits the result of a previous computation, the pipeline may have to stall until the result is available. A (conditional) branch instruction is time consuming. If an instruction awaits the result of a previous computation, the pipeline may have to stall until the result becomes available.

If an instruction creates a value, there may be a latency that has to elapse before another instruction can use this value. Careful instruction scheduling is required. The pipeline structure of modern architectures requires RISC Architectures.
Instruction Scheduling

Purpose:
- minimizes stalls and delays.
- Fill delay slots with useful computations, minimize execution time of basic blocks.

Tool:
- Dependency analysis.
- Uncover legal reordering.

Applications:
- Filling delay slots is important for all programs.
- Beyond computations, available parallelism in basic blocks.

Honors Compilers, NYU, Spring, 2007
Dependence Relations

A data dependence is a constraint that arises from the flow of data between statements. Violating a data dependence by reordering may lead to incorrect results.

- If both $S_1$ and $S_2$ read the value of some variable, there is an input dependence between them. This does not impose an ordering.
- If both $S_1$ and $S_2$ set the value of some variable, there is an output dependence between them. This does not impose an ordering, but implies that these two instructions cannot be executed in parallel.
- If $S_1$ uses some variable's value and $S_2$ sets it, there is an anti-dependence between them.
- If $S_1$ uses some variable's value and $S_2$ sets it, there is an input dependence between $S_1$ and $S_2$.
- If $S_1$ uses a value that $S_2$ uses, this is flow dependence or true dependence between statements. Reordering a data dependence by reordering may lead to incorrect results.
The Dependence DAG of a Basic Block

- There is an edge connecting \( I_1 \) to a later \( I_2 \) if:
  - \( I_1 \) writes a register or location that \( I_2 \) uses:
  - \( I_2 \) modifies:
  - \( I_1 \) and \( I_2 \) write to the same register or location:
  - \( I_1 \) and \( I_2 \) exhibit a structural hazard: a load followed by a store cannot be interchanged unless the addresses are known to be distinct:
    \[ X := A[i]; \]
    \[ A[i] := Y; \]
  - Cannot interchange. \( X \) might get \( Y \)

- If there is an edge between \( I_1 \) and \( I_2 \), \( I_2 \) must not start executing until \( I_1 \) has executed for some number of cycles.
Example

1: \[ R_3 = [R_{15}] \]
2: \[ R_4 = [R_{15}+4] \]
3: \[ R_2 = R_3 - R_4 \]
4: \[ R_5 = [R_{12}] \]
5: \[ R_{12} = R_{12} + 4 \]
6: \[ R_6 = R_3 + R_5 \]
7: \[ R_7 = [R_{15}+4] \]
8: \[ R_8 = R_6 + 2 \]

- needs \( R_4 \); stall one cycle
Edges in dependency graph are labeled with latencies (\( \geq 1 \)).

Conflict depends on relative starting time of two instructions.


"Add" uses unpack 

U: Unpack 

S: Operand shift 

R: Adder round 

N: Multiplier second stage 

M: Multiplier first stage 

E: Exception test 

A: Mantissa add 

E.g.: Floating-point unit on MIPS:

Functional unit is pipelined, consists of multiple resources.

Contention for Resources
Important use of dependency graph: fills delay slots

Branch Scheduling

R4 := R2 + R3
R5 := R2 - 1
go to L1
R5 := R2 - 1 (stall)
go to L1
R2 := [R1]
R3 := [R1 + 4]

nop
R4 := R2 + R3
R3 := [R1 + 4]
R2 := [R1]

(b) Branch takes two cycles to reach destination
Conditional Jumps and Delay Slots

Instruction in delay slot is executed while jump is in progress.

What if jump is not taken?

Need mechanism to annul instruction.

Branch prediction: Assume that a backwards conditional jump is usually taken. Move first instruction in delay slot to loop start.

Good heuristics for loops: Assume that a backwards conditional jump is usually taken. Move first instruction in delay slot to loop start.

If both destinations start with same instruction, ideal choice for delay slot.

Good heuristics for loops: Assume that a backwards conditional jump is usually taken. Move first instruction in loop to delay slot.

Delay prediction: Assume target is known. Fill delay slot with first instruction in target block.

Push.

Call instruction has delay slot filled with parameter.

In loop to delay slot for branch at end.
Finding optimal schedule for DAG is NP-complete

For leaf: execution time of instruction

For inner node: maximum delay imposed by successors

\[ T = L - 6 \]

is a latency of 2 between \( u \) and \( w \) cannot start later than \( T \), and therefore \( T \) is followed by \( u \), \( w \) can start at \( T \), and there

E.g. if \( I_1 \) is followed by \( I_2 \) and \( I_3 \), then \( I_1 \) cannot start later than \( T \) – 4

Simple algorithm is \( O(N^2) \) at worst, usually linear

Finding optimal schedule for DAG is NP-complete

A Greedy Algorithm: List Scheduling

Roots of DAG are instructions without predecessors

First pass: from leaves to roots: compute latest

Possible starting time for each instruction to end of block

A.Pnueli

AGreedyAlgorithm:ListScheduling

Finding optimal schedule for DAG is NP-complete

Honors Compilers, NYU, Spring, 2007
Second Pass: Schedule

List Scheduling: Second Pass

At each step:

- Choose instruction that frees register
- Choose instruction that uses least used pipeline, or
- Choose earliest starting time, or
- Use heuristics:
  - Choose from D1 if unique, else from D2 if unique, else
  - from starting time of their predecessors

D1. Candidates with the largest remaining delay
D2. Candidates with the largest remaining slack
D3. Candidates with the earliest possible start time (computed)

Lecture 11: Architecture-Dependent Optimization

A. Pnueli

Honors Compilers, NYU, Spring, 2007

280
Is harder:

Possible disadvantages: code size increases, debugging

Both cases it can enable other optimizations.

Can be done at the tree level or at the RTL level. In

It saves the cost of the call
It creates larger basic blocks
It exposes the values of the actuals in the body

Inlining subprogram bodies is often very effective:

Calls make optimizations harder. There is a large

Procedure Integration: Inlining
Inlining as a Tree Transformation

- Introduce temporary to hold return value of function
- Replace multiple return statements where needed
- Inlining works like instantiation: replace formals with actuals, complete analysis and expansion of inserted body
- Global references are captured at the point of definition
- Each inlined body needs its own local variables
- Treat body of subprogram as a generic unit

A. Pnueli
Name Capture: Recognize Global Entities

A. Pnueli

Lecture 11: Architecture-Dependent Optimization

function memo(x: integer)
return integer
is
local: integer := x
2;

begin
Saved := Saved + local + 15;

-- Saved in the same entity in all inlinings
-- Saved = Saved + local + 15;

return Saved;
end

Val := memo(15);

-- maybe superfluous if context is assignment
-- set of local variables (and "result")
-- each inlining has its own
declare
local: integer := 15 * 2;

begin
Saved := Saved + local + 15;

result := Saved;

end

Val := result

-- Saved in the same entity in all inlinings
-- Saved = Saved + local + x;

Honors Compilers, NYU, Spring, 2007
Handling Return Statements

Subprograms need a label to serve as a single exit point.

In a function:
1. Identify target of result, or create temporary.
2. Replace return with assignment to target, followed by a goto to exit label.

Optimizations:

- If procedure has no return statement, exit label is superfluous.
- If procedure has no return, can replace right-hand side with expression.
- If body of function is single return statement and context is
  Optimizations:
- In a procedure: replace return with goto to exit label.

Honors Compilers, NYU, Spring 2007
Parameter Passing

If actual is an expression, it is evaluated once: create temporary in block and replace formal with temporary.
Parameter Passing: Variables

An in-out parameter cannot create a temporary for it; must use a renaming declaration.

procedure incr(x: in-out integer) is
begin
x := x + 1;
end;

becomes

\[
\begin{align*}
\text{procedure incr(a(i)) is} \\
\text{begin} \\
\text{c1 := c1 + 1;} \\
\text{end;}
\end{align*}
\]

\[
\begin{align*}
\text{An in-out parameter is a location, cannot create a} \\
\text{temporary for it; must use a renaming declaration.}
\end{align*}
\]
Inlining, must be applied when analyzing inlined block.

- Status of constraint checks is part of closure of body to

- It suppressed at the point of call

- If suppressed in the inlined block, even

- If constraint checks are not suppressed in the body,'

- Program

- Semantics of inlined call must be identical to original

Context includes more than global names
Specialized Inlining: Loop Unrolling

for k in 1..N/r
  loop
    loop-body[j]
  endloop;

for k in N/r+1..N
  loop
    loop-body end loop;

end loop;

for k in N/r+1..N
  loop
    loop-body end loop;
  endloop;

end loop;

\[ j \rightarrow j+r-1 \]

\[ \ldots \]

\[ \ldots \]

\[ \ldots \]

for k in 1..N/r
  loop
    loop-body[j]
  endloop;

for k in 1..N/r
  loop
    loop-body[j]
  endloop;

end loop;

for k in N/r+1..N
  loop
    loop-body[j]
  endloop;

end loop;

for k in N/r+1..N
  loop
    loop-body[j]
  endloop;

end loop;

Honors Compilers, NYU, Spring 2007

A. Pnueli