Architecture-Dependent Optimization

Functional Units, delay slots, and dependency analysis
The pipelined structure of modern architectures requires careful instructions scheduling. 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. If an instruction creates a value, there may be a latency that has to elapse before another instruction can use this value. The pipeline structure of modern architectures requires RISC Architectures.
Lecture 11: Architecture-Dependent Optimization

**Purpose:**

- Instruction Scheduling

  - minimizes stalls and delays.
  - Fill delayslots with useful computations, minimize execution time of basic blocks.

**Tool:**

- Dependency analysis
  - Uncover legal reordering

**Applications:**

- Filling delayslots is important for all programs
- Dependency analysis is critical for reordering of loops
- Applications beyond
  - Computations on vector processors and others
  - Available parallelism in basic blocks and beyond

**Tool:**

- Basic block
- maximize execution time of useful computations
- Fill delayslots and delays.

**Instruction Scheduling**

 compromisers, NYU, Fall, 2009

Honors Compilers, NYU, Fall, 2007

A. Prueil
A.Pnueli

Dependence Relations

- 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 parallel dependence between them.

- If both $S_1$ and $S_2$ set the value of some variable, there is an output dependence between them.

- If $S_1$ uses some variable's value and $S_2$ sets it, there is an anti-dependence between them.

- If $S_1$ sets a value that $S_2$ uses, this is flow dependence or true dependence between them.

A data dependence is a constraint that arises from the flow of data.
Lecture 11: Architecture-Dependent Optimization

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 $I_2$ modifies.
- $I_1$ and $I_2$ write to the same register or location:
  - $I_1$ uses a register or location that $I_2$ modifies.
  - $I_2$ uses a register or location that $I_1$ modifies.
- $I_2$ exhibits a structural hazard: a load followed by a store.

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.

If there is an edge connecting $I_1$ to a later $I_2$, $I_1$ might get $X$ and $I_2$ might get $Y$.

---

A.Pnueli

Honors Compilers, NYU, Fall, 2007

280
Example

1. $R3 := [R15]$
2. $R4 := [R15 + 4]$
3. $R2 := R3 - R4$
4. $R5 := [R12]$
5. $R12 := R12 + 4$
6. $R6 := R3 + R5$
7. $[R15 + 4] := R3$
8. $R5 := R6 + 2$

- needs $R4$; stall one cycle
Contention for Resources

Architecture-Dependent Optimization

Contdlict depends on relative starting time of two instructions.


Instructions through the pipeline may conflict on use of resources.

Functional unit is pipelined, consists of multiple resources.

Edges in dependency graph are labeled with latencies (≥ 1).

Honors Compilers, NYU, Fall, 2007

A. Pnueli

Lecture 1: Architecture-Dependent Optimization
Important use of dependency graphs: fill delay slots.

Branch Scheduling

\[
\begin{align*}
    R_4 &= R_2 + R_3 \\
    \text{goto L1} \\
    R_5 &= R_2 - 1 \\
    R_3 &= [R_1 + 4] \\
    R_2 &= [R_1] \\
\end{align*}
\]

(b) Branch takes two cycles to reach destination.

A. Pnueli

Honors Compilers, NYU, Fall 2007

Lecture 11: Architecture-Dependent Optimization

Branch Scheduling
Conditional Jumps and Delay Slots

- Call instruction has delay slot filled with parameter push.
- Branch prediction: assume that a backwards conditional
  jump is usually taken. Move first instruction in loop to delay slot
  for loops.
- Good heuristics for loops: if both destinations start with same
  instruction, ideal choice for delay slot.
- Instruction in target block: assume target is known, fill
delay slot with first instruction.
- Branch prediction: what if jump is not taken?
- Need mechanism to annul instruction.
- Instruction in delay slot is executed while instruction
  in progress.

Lecture 11: Architecture-Dependent Optimization
A Greedy Algorithm: List Scheduling

Finding optimal schedule for DAG is NP-complete

For inner node: maximum delay imposed by successors

For leaf: execution time of instruction

Roots of DAG are instructions without predecessors

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

Finding optimal schedule for DAG is NP-complete

E.g. If $I_1$ is followed by $I_2$, $I_2$ can start at $T = I_1 + 2$, and there is a latency of 2 between $I_1$ and $I_2$, and then $I_1$ cannot start later than $T$.

For leaf $I_n$: execution time of instruction

For leaf $I_n$: execution time of instruction

For leaf $I_n$: execution time of instruction

For leaf $I_n$: execution time of instruction

For leaf $I_n$: execution time of instruction

First pass: from leaves to roots: compute latest possible starting time for each instruction to end of block

Roots of DAG are instructions without predecessors

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

Finding optimal schedule for DAG is NP-complete

A Greedy Algorithm: List Scheduling

For inner node: maximum delay imposed by successors

For leaf: execution time of instruction

Roots of DAG are instructions without predecessors

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

Finding optimal schedule for DAG is NP-complete

A Greedy Algorithm: List Scheduling
A. Pnueli

List Scheduling: Second Pass

Second pass: from roots to leaves; schedule instructions that can start as early as possible from now.

Honors Compilers, NYU, Fall, 2007

286
Procedure Integration: Inlining

Possible disadvantages: code size increases, debugging is harder.

- Possible disadvantages: code size increases, debugging is harder.
  - Both cases it can enable other optimizations.
  - Can be done at the tree level or at the RTL level.
  - If 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 payoff to local optimizations over large basic blocks.

A. Pnueli

Lecture 11: Architecture-Dependent Optimization

Honors Compilers, NYU, Fall, 2007
Inlining as a Tree Transformation

A. Pnueli

Lecture 11: Architecture-Dependent Optimization

- Introduce temporary to hold return value of function
- Replace multiple return statements where needed
- 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

Inlining works like instantiation: replace formals with actuals, complete analysis and expansion of inserted body

Replacemultiplereturnstatementswhereneeded
Introducetemporarytoholdreturnvalueoffunction
Replacemultiplereturnstatementswhereneeded
Body
Globalreferencesarecapturedatthepointofdefinition
Eachinlinedbodyneedsthisownlocalvariables
Treatbodyofsubprogramasanagenericunit
Name Capture: Recognize Global Entities

\begin{verbatim}
A.Pnueli

function memo(x: integer) return integer is
  local: integer := x
  local := local + x; -- Saved in the same entity in all inlinings
  Saved := Saved + local + x;
  return Saved;
end memo;

Val := memo(15);

Val := result
end;

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

Val := memo(15);

\end{verbatim}
Handling Return Statements

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

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

In a procedure:
- Replace return with goto to exit label.

Optimizations:
- If procedure has no return statement, exit label is superfluous.
- If procedure has return statement, can replace right-hand side with expression assignment.
- If body of function is single return statement and context is assignment:
  - Replace return with assignment to temporary for it.

Honors Compilers, NYU, Fall, 2007
If actual is an expression, it is evaluated once; create temporary in block and replace formal with temporary

### Parameter Passing

- Context is assignment

\[ \text{Val2} := \text{memo}(x^{f(x)}) \]
Parameter Passing: Variables

A parameter is an object that is placed in a memory location called a parameter location.

- **An in-out parameter** is a location that cannot be created.
- A temporary for it must use a renaming declaration.

### Parameter Passing

#### An in-out parameter

```plaintext
procedure incr(x: integer)
begin
    x := x + 1;
end;
```

#### Becomes

```plaintext
declare c1: integer renames a(i);
begin
    c1 := c1 + 1;
end;
```

### Parameter Passing

#### An in-out parameter

```plaintext
procedure incr(a(i));
...
begin
    x := x + 1;
end;
```
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

  they must not be suppressed in the inlined block, even if suppressed at the point of call

- It constraint checks are not suppressed in the body,

  program

  Semantics of inlined call must be identical to original

Context includes More than Global Names

A. Pnueli
Specialized Inlining: Loop Unrolling

for \( j \in \{1, 1+N, \ldots, 1+N/r\} \)
end loop;

\[ j := j + r; \]

Loop-body[\( j + 1 \)] \( \leftarrow \) Loop-body[\( j + r - 1 \)];

\( j := j + r; \)

\( \ldots \)

for \( k \in \{1+N/r+1, \ldots, N\} \)
end loop;

Honors Compilers, NYU, Fall 2007