Register-Aware Pre-Scheduling

- High instruction level parallelism does not help if it comes at the price of high register pressure → much spill code inserted afterwards
- Register allocator complexity grows with interfering live ranges

Idea:
- Avoid spilling in graph-coloring register allocation
  - Limit register pressure to #physical registers
  - Reordering/linearization to reduce overlap of live ranges
- Register-aware Pre-Scheduling:
  - Add further ordering constraints (artificial data dependences) to the partial order (dataflow graph, tree or DAG) given by data dependences → linear order, restricts instruction scheduler
  - Goal: Trade reduced instruction level parallelism for reduced spill code

Spill Code is Undesirable

- Spill code = memory accesses (stores, loads)
  - Time + energy penalty
- Even if the target processor internally has many more registers than addressable with the instructions' register field bits and can rename registers on-the-fly, spill code cannot be recognized as such and removed
- Even if the processor has many registers available, live registers need to be saved+restored at function calls

Simple Code Generation Algorithm

ALSU2e Ch. 8.6
for Self-Study

Standard topic in Compiler Construction I
– Not presented here

Simple code generation algorithm (1)

- Input: Basic block graph (LIR/quadruples grouped in BB’s)

Principle:
- Keep a (computed) value in a register as long as possible, and move it to memory only
  1. if the register is needed for another calculation
  2. at the end of a basic block.

A variable x is used locally after a point p if x’s value is used within the block after p before an assignment to x (if any) is made.

All variables (except temporaries) are assumed to be alive (may be used later before possibly being reassigned) after a basic block.

"is used locally" and "alive"

"alive variables" is a backward data-flow analysis problem.
Simple code generation algorithm (2)

reg(R): current content (variable) stored in register R
adr(A): list of addresses ("home" memory location, register) where the current value of variable A resides

Generate code for a quadruple \( Q = (op, B, C, A) \) (op a binary oper.):

- If \( reg(RB) \neq B \), generate `LOAD B, RB; reg(RB) \leftarrow B; adr(B) \leftarrow adr(B) \cup \{RB\}`
- If \( reg(RC) \neq C \), generate `LOAD C, RC; reg(RC) \leftarrow C; adr(C) \leftarrow adr(C) \cup \{RC\}`
- Generate `OP RB, RC, RA` (where `OP` implements op)
- `adr(A) \leftarrow \{RA\};` // old value of A in memory is now stale
- `reg(RA) \leftarrow A;`
- If \( B \) and/or \( C \) no longer used locally and are not alive after the current basic block, free their registers `RB, RC` (update `reg, adr)`

After all quadruples in the basic block have been processed, generate `STORE`s for all non-temporary var's that only reside in a register.

Simple code generation algorithm (3)

getreg( quadruple \( Q = (op, B, C, A) \) ) determines (RB, RC, RA):

- Determine RB:
  - If \( adr(B) \) contains a register \( RB \) and \( reg(RB) = B \), then use RB.
  - If \( adr(B) \) contains a register \( RB \) with \( reg(RB) \neq B \) or \( adr(B) \) contains no register at all:
    - Use an empty register as \( RB \) if one is free.
    - If no register is free: Select any victim register \( R \) that does not hold \( C \).
    - \( v \) \leftarrow \{ v: R \in adr(V) \} \) (there may be several \( v \)'s, due to control flow)
      - If for all \( v \in V \), \( adr(v) \) contains some other place beyond \( R \):
        - OK, use \( R \) for \( RB \).
      - If \( C \in V \), retry with another register \( R \) instead.
      - If \( A \in V \), OK, use \( RA = RA \) for \( RB \).
- Determine RC: similarly
- Prefer candidates \( R \) that require least spills
- Determine RA: similarly, where:
  - Any \( R \) with \( reg(R) = \{A\} \) can be reused as RA
  - If \( B \) is not used after \( Q \), and \( reg(RB) = \{B\} \), can use \( RA = RB \).
  - (similarly for \( C \) and \( RC \))

Example

Generate code for this basic block in pseudo-quadruple notation:

\[
\begin{align*}
T1 &:= a + b; \\
T2 &:= c + d; \\
T3 &:= e + f; \\
T4 &:= T2 \cdot T3; \\
g &:= T1 - T4;
\end{align*}
\]

Initially, no register is used.

Assume \( a, b, c, d, e, f, g \) are alive after the basic block, but the temporaries are not.

Machine model: as above, but only 3 registers \( R0, R1, R2 \)

(see whiteboard)

Solution

\( NB \) – several possibilities, dep. on victim selection

1. LOAD a, R0 \hspace{1cm} \text{ // now } \text{adr}(a) = \{A, R0, RA\}, \text{reg}(R0) = \{a\}
2. LOAD b, R1
3. ADD R0, R1, R2 \hspace{1cm} \text{ // reuse } R0, R1 \text{ for } c, d, \text{ as } a, b \text{ still reside in memory}
   \text{ // use } R0 \text{ for } T2, \text{ as } c \text{ still available in memory.}
4. LOAD c, R0
5. LOAD d, R1
6. ADD R0, R1, R0 \hspace{1cm} \text{ // now } \text{adr}(T2) = \{R0, RA\}, \text{reg}(R0) = \{T2\}
   \text{ // reuse } R1 \text{ for } e, \text{ need a register for } f \text{ – none free! Pick victim } \text{R0}
7. STORE R0, 12(fp) \hspace{1cm} \text{ // spill } R0 \text{ to a stack location for } T2, \text{ e.g. at } fp+12
8. LOAD e, R1
9. LOAD f, R0
10. ADD R1, R0, R1 \hspace{1cm} \text{ // now } \text{adr}(T3) = \{R1\}, \text{reg}(R1) = \{T3\}
11. LOAD T2, R0 \hspace{1cm} \text{ // reload } T2 \text{ to } R0
12. MUL R0, R1, R0 \hspace{1cm} \text{ // } T4 \text{ in } R0
13. SUB R2, R0, R2 \hspace{1cm} \text{ // } g \text{ in } R2
14. STORE R2, g

14 instructions, including 9 memory accesses (2 due to spilling)
Example – slightly reordered

Generate code for this basic block in pseudo-quadruple notation:

\[
\begin{align*}
T2 &\leftarrow a + b; \\
T3 &\leftarrow a + c; \\
T4 &\leftarrow T2 * T3; \\
T1 &\leftarrow a + b; \\
g &\leftarrow T1 - T4; \\
\end{align*}
\]

Initially, no register is used.

Assume a, b, c, d, e, f, g are alive after the basic block, but the temporaries are not.

Machine model: as above, but only 3 registers R0, R1, R2

(see whiteboard)

Solution for reordered example

1. LOAD c, R0
2. LOAD d, R1
3. ADD R0, R1, R2  // now addr(T2)=<R2>, reg(R2)=T2
    // reuse R0 for e, R1 for f
4. LOAD e, R0
5. LOAD f, R1
6. ADD R0, R1, R0  // now addr(T3)=<R0>, reg(R0)=T3
    // reuse R0 for T4
7. MUL R0, R1, R0  // now addr(T4)=<R0>, reg(R0)=T4
    // reuse R1 for a, R2 for b, R1 for T1
8. LOAD a, R1
9. LOAD b, R2
10. ADD R1, R2, R1   // now addr(T1)=<R1>, reg(R1)=<T1>
    // reuse R1 for g
11. SUB R1, R0, R1   // g in R1
12. STORE R1, g

There are 12 instructions, including 7 memory accesses.
No spilling! Why?

Explanation

- Consider the data flow graph (here, an expression tree) of the example:

  ![Data Flow Graph](image)

  - T1 := a + b;
  - T2 := c + d;
  - T3 := e + f;
  - T4 := T2 * T3;
  - T1 := a + b;
  - g := T1 - T4;

  Moving T1 := a + b; here does not modify the semantics of the code. (Why?)

- Idea:
  For each subtree T(v) rooted at node v:
  How many registers do we need (at least) to compute T(v) without spilling?

  Call this number label(v) (a.k.a. "Ershov numbers")

  If possible, at any v, code for "heavier" subtree of v (higher label) should come first.

Labeling algorithm [Ershov 1958]

- Yields space-optimal code (proof: [Sethi, Ullman 1970]) (using a minimum number of registers). (proof: [Sethi, Ullman 1970])
- (Time is fixed as no spill code is generated.)
- The problem is NP-complete for expression DAGs [Sethi'75]
- Solutions for DAGs: [K. '95], [K. '98], [Amaral et al.'00]
- If #machine registers exceeded: Spill code could be inserted afterwards. (Why?)

2 phases:

- Labeling phase
  - bottom-up traversal of the tree
  - recursively generating code for each node v

- Code generation phase
  - top-down traversal of the tree
  - semi-optimal code generation for heavier subtree first

Labeling phase

Bottom-up, calculate the register need for each subtree T(v):

- If v is a leaf node, label(v) \( \leftarrow 1 \)
- If v is a unary operation with operand node v1, label(v) \( \leftarrow \) label(v1)
- If v is a binary operation with operand nodes v1, v2:
  - \( m \leftarrow \max(\text{label}(v1), \text{label}(v2)) \)
  - If \( \text{label}(v1) = \text{label}(v2) \) then \( \text{label}(v) \leftarrow m + 1 \)
  - Else \( \text{label}(v) \leftarrow m \)

Space – Optimal Pre-Scheduling for Trees

The Labeling Algorithm

Labeling algorithm [Ershov 1958]

- Yields space-optimal code (proof: [Sethi, Ullman 1970])
- (Using a minimum number of registers)
- (Without spilling)
- (Or at least)
- (Min. number of stack locations)
- (Why?)

- The problem is NP-complete for expression DAGs [Sethi'75]
- Solutions for DAGs: [K. '95], [K. '98], [Amaral et al.'00]
- If #machine registers exceeded: Spill code could be inserted afterwards. (Why?)

2 phases:

- Labeling phase
  - Bottom-up traversal of the tree
  - Recursively generating code for each node v

- Code generation phase
  - Top-down traversal of the tree
  - Semi-optimal code generation for heavier subtree first
Example – labeling phase

Code generation phase

- Register stack freeregs of currently free registers, initially full
- Register assignment function reg from values to registers, initially empty

```c
// generate space-opt. code for subtree T(v)
gencode(v) {
    R freeregs.pop();
    reg(v) = R;  
    generate(LOAD v, R);
    if (v is a leaf node):
        1. R freeregs.pop();
           reg(v) = R;
           generate(LABEL v, R);
        2. generate(OP R1, R1);     
           reg(v) = R1;
        3. if (label(v1) >= label(v2))
            gencode(v1);
            gencode(v2);
        else
            gencode(v2);
            gencode(v1);
    else
        // code for T(v2) first:
        gencode(v2);
        gencode(v1);
    freeregs.push(R2);  // return register R2, keep R1 for v
}
```

Remarks on the Labeling Algorithm

- Still one-to-one or one-to-many translation from quadruple operators to target instructions
- The code generated by gencode() is contiguous
  (a subtree’s code is never interleaved with a sibling subtree’s code).
  - E.g., code for a unary operation v immediately follows the code for its child v1.
  - Good for space usage, but bad for execution time on pipelined processors!
  - Space-optimal (non-contiguous) ordering for DAGs can be computed by a dynamic programming algorithm [K. 1998]
  - There are expression DAGs for which a non-contiguous ordering exists that uses fewer registers than any contiguous ordering. [K., Rauber 1995]
- The labeling algorithm can serve as a heuristic (but not as optimal algorithm) for DAGs if gencode() is called for common subexpressions only at the first time.

References