Register-Aware Pre-Scheduling

Christoph Kessler, IDA,
Linköpings universitet, 2009.

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

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
    - Pre-spilling
- **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

Simple Code Generation Algorithm

ALSU2e Ch. 8.6

for Self-Study

**Standard topic in Compiler Construction I
– Not presented here**

Christoph Kessler, IDA,
Linköpings universitet, 2009.

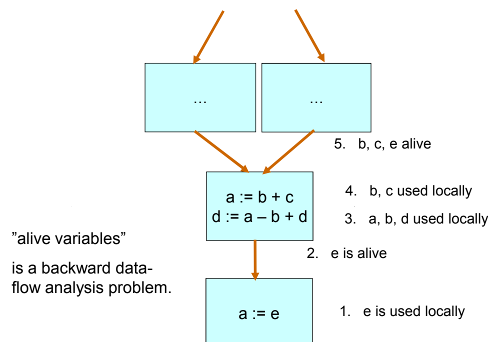
Simple code generation algorithm (1)

- 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.

BB3:
(ADD, a, b, **x**)
…
(MUL, **x**, y, t1)
…
(ASGN, t1, 0, **x**)
…

- 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

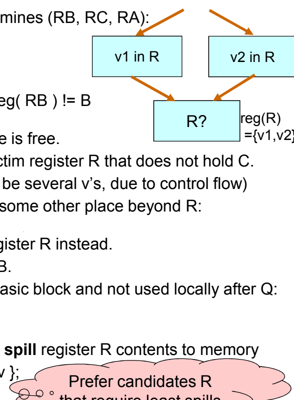
Generate code for a quadruple Q = (op, B, C, A): (op a binary oper.)

- (RB, RC, RA) ← getreg(Q); // selects registers for B, C, A – see later
- If reg(RB) != B
  generate **LOAD** B, RB; reg(RB) ← B; adr(B) ← adr(B) U { RB }
- If reg(RC) != C
  generate **LOAD** C, RC; reg(RC) ← C; adr(C) ← adr(C) U { RC }
- generate **OP** RB, RC, RA (where **OP** implements op)
  adr(A) ← { RA }; // old value of A in memory is now stale
  reg(RA) ← 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) != 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 ← { 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 R=RA for RB.
      - If all v in V not alive after this basic block and not used locally after Q: OK, use R for RB
      - Otherwise: for each v in V,
        - generate **STORE** R, v; // **spill** register R contents to memory
        - adr(v) = adr(v) – { R } U { &v };
- Determine RC: similarly

Prefer candidates R that require least spills

Simple code generation algorithm (4)

- 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

Christoph Kessler, IDA, Linköpings universitet, 2009.

Example

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

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

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 // now adr(a) = { &a, R0 }, reg(R0)={a}
2. LOAD b, R1
3. ADD R0, R1, R2 // now adr(T1) = {R2}, reg(R2)={T1}
   // reuse R0, R1 for c, d, as a, b still reside in memory
   // use R0 for T2, as c still available in memory.
4. LOAD c, R0
5. LOAD d, R1
6. ADD R0, R1, R0 // now adr(T2) = {R0}, reg(R0)={T2}
   // reuse R1 for e, need a register for f – none free! Pick victim R0
7. STORE R0, 12(fp) // **spill** R0 to a stack location for T2, e.g. at fp+12
8. LOAD e, R1
9. LOAD f, R0
10. ADD R1, R0, R1 // now adr(T3) = {R1}, reg(R1)={T3}
11. LOAD T2, R0 // reload T2 to R0
12. MUL R0, R1, R0 // T4 in R0
13. SUB R2, R0, R2 // g in R2
14. STORE R2, g

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

14 instructions,
including 9 memory accesses
(2 due to spilling)

Example – slightly reordered

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

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?)

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

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

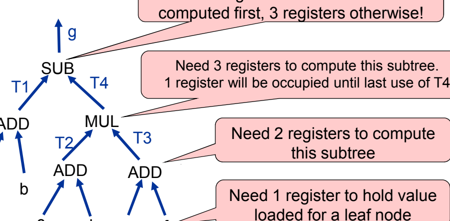
1. LOAD c, R0
2. LOAD d, R1
3. ADD R0, R1, R2 // now adr(T2)={R2}, reg(R2)={T2}
   // reuse R0 for e, R1 for f:
4. LOAD e, R0
5. LOAD f, R1
6. ADD R0, R1, R0 // now adr(T3) = {R0}, reg(R0)={T3}
   // reuse R0 for T4:
7. MUL R0, R1, R0 // now adr(T4)={R0}, reg(R0)={T4}
   // reuse R1 for a, R2 for b,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

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

Explanation

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

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



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.

TDDC86 Compilers Optimizations and Code Generation

Space – Optimal Pre-Scheduling for Trees

The Labeling Algorithm

Christoph Kessler, IDA, Linköpings universitet, 2009.

Labeling algorithm [Ershov 1958]

- Yields **space-optimal code** (proof: [Sethi, Ullman 1970]) (using a minimum #registers **without spilling**, or min. # stack locations) for expression **trees**.
  - (Time is fixed as no spill code is generated.)
- The problem is NP-complete for expression DAGs! [Sethi'75]
  - Solutions for DAGs: [K., Rauber '95], [K. '98], [Amaral et al.'00]
- If #machine registers exceeded: Spill code could be inserted afterwards for excess registers, but not necessarily (time-) optimal then…

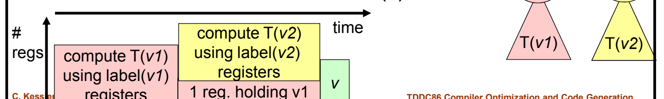
2 phases:

- **Labeling phase**
  - bottom-up traversal of the tree
  - computes label(v) recursively for each node v
- **Code generation phase**
  - top-down traversal of the tree
  - recursively generating code for heavier subtree first

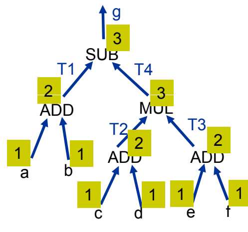
Labeling phase

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

- **If** *v* is a leaf node,
  label(*v*) ← 1
- **If** *v* is a unary operation with operand node v1,
  label(*v*) ← label(*v1*)
- **If** *v* is a binary operation with operand nodes v1, v2:
  m ← **max**(label(*v1*), label(*v2*));
  if (label(*v1*) = label(*v2*)) label(*v*) ← *m* + 1;
  else label(*v*) ← *m*;



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

gencode(*v*) { // generate space-opt. code for subtree T(v)

- **If** *v* is a leaf node:
  1. *R* ← freeregs.pop(); reg(*v*) ← *R*; // get a free register R for v
  2. generate(`LOAD` *v*, *R*);
- **If** v is a unary node with operand v1 in a register reg(*v1*)=*R1*:
  1. generate(`OP` *R1*, *R1*); reg(*v*) ← *R1*;
- **If** v is a binary node with operands *v1, v2* in reg(*v1*)=*R1*, reg(*v2*)=*R2*:
  1. **if** (label(*v1*) >= label(*v2*)) // code for T(*v1*) first:
gencode(*v1*);
gencode(*v2*);
else // code for T(*v2*) first:
gencode(*v2*);
gencode(*v1*);
  2. generate(`OP` *R1, R2, R1*);
  3. 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

- R. Sethi, J. Ullman: The generation of optimal code for arithmetic expressions. *J. of the ACM* **17**:715-728, 1970
- R. Sethi: Complete Register Allocation Problems. *SIAM J. Comput.* **4**:226-248, 1975
- C. Kessler, T. Rauber: Generating Optimal Contiguous Evaluations for Expression DAGs. *Computer Languages* **21**(2), 1995.
- C. Kessler: Scheduling Expression DAGs for Minimal Register Need. *Computer Languages* **24**(1), pp. 33-53, 1998.