FACULTY
OF INFORMATICS
Masaryk University

PA039: Supercomputer Architecture and Intensive Computing

Compiling and Code Optimization

Luděk Matyska

Spring 2024

Repetition – RISC processors

- Limited number of instructions, same size
- Simple address modes, Load/Store, sufficient number of registers
- Delayed branches, branch prediction, out-of-order execution
- Superscalar (e.g. 2xFPU, 2xALU, special address instructions)
- Superpipeline
- Caches

Optimizing Compiler

- Translation to the **Intermediate language**
- Optimization
  - intra-procedural analysis
  - cycle optimization
  - global optimization (inter-process optimization)
- Code generation
  - use of all superscalar units

Intermediate Language

- Quadruple (generally ***n***-tuple)
  - Instruction: operator, two operands, result
  - Example
    - Operation op writen as: `X := Y op Z`
- Memory: accessible through temporary variables `t`***n***
- Branches: condition calculated separately
- Branches: jumps to absolute addresses

Basic translation

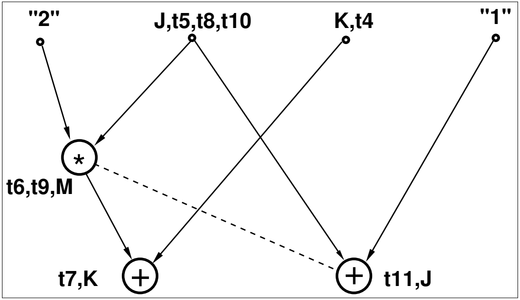
```
while ( j < n ) {
          k = k + j*2
          m = j*2
          j++
 }
A::   t1  := j
      t2  := n
      t3  := t1 < t2
      jmp (B) t3
      jmp (C) TRUE
B::   t4  := k
      t5  := j
      t6  := t5*2
      t7  := t4+t6
      k   := t7
      t8  := j
      t9  := t8*2
      m   := t9
      t10 := j
      t11 := t10+1
      j   := t11
      jmp (A) TRUE
C::
```

Basic blocks

- Program is represented as a *flow graph*
- Block – a code segment without branches/jumps
  - One **entry** and one **exit** point
  - Block as a DAG (Directed Acyclic Graph)
- Optimization within blocks
  - Removal of repeated (sub)expressions
  - Removal of redundant variables

Directed Acyclic Graph

```
B::  t4  := k
     t5  := j
     t6  := t5*2
     t7  := t4+t6
     k   := t7
     t8  := j
     t9  := t8*2
     m   := t9
     t10 := j
     t11 := t10+1
     j   := t11
     jmp (A) TRUE
```



Modified translation

```
B::  t4   := k
     t5   := j
     t6   := t5*2
     t7   := t4+t6
     k    := t7
     t8   := j
     t9   := t8*2
     m    := t9
     t10  := j
     t11  := t10+1
     j    := t11
     jmp (A) TRUE
```

```
B::  t4   := k
     t5   := j
     t6   := t5*2
     m    := t6
     t7   := t6+t4
     k    := t7
     t11  := t5+1
     j    := t11
     jmp (A) TRUE
```

Additional concepts

- *Variables*
  - Definition and place of use
- *Cycles*
- *Target code generation*
  - Includes the so-called peephole optimization

Optimized code

```
A::   t1   := j
      t2   := n
      t3   := t1 < t2
      jmp (B) t3
      jmp (C) TRUE
B::   t4   := k
      t5   := j
      t6   := t5*2
      t7   := t4+t6
      k    := t7
      t8   := j
      t9   := t8*2
      m    := t9
      t10  := j
      t11  := t10+1
      j    := t11
      jmp (A) TRUE
C::
```

```
A::   t1   := j
      t2   := n
      t4   := k
      t9   := m
      t12  := t1+t1
      t3   := t1 >= t2
      jmp (B1) t3
B::   t4   := t4+t12
      t9   := t12
      t1   := t1+1
      t12  := t12+2
      t3   := t1 < t2
      jmp (B) t3
B1::  k    := t4
      m    := t9
C::
```

Classical optimizations

- Copy propagation
  - Examples:

```
X = Y                      X = Y
Z = 1. + X       ⟹         Z = 1. + Y
```

- Constants processing
  - constants propagation
  - constant folding
- Dead-code elimination
  - inaccessible code
  - saving cache capacity for instructions

Classical optimizations II

- Strength reduction
  - Example: K**2 $\Longrightarrow$ K*K
- Variable renaming
  - Example

```
x = y*z;                 x0 = y*z;
q = r+x+x;      ==>      q  = r+x0+x0;
x = a+b                  x  = a+b
```

- Common subexpressions elimination
  (important especially for evaluation of array indices)

Classical optimizations III

- Move of invariant code from cycles
- Simplification of induction variables (expressions with them)
  - `A(I)` is usually computed as:
    `address = base_address(A) + (I-1)*sizeof_datatype(A)`
    which can be in a linear cycle easily simplified to
    outside cycle:
    `address = base_address(A) - sizeof_datatype(A)`
    within cycle:
    `address = address + sizeof_datatype(A)`
- Register allocation

Garbage elimination

- Procedures, macros
  - Inlining
- Conditional expressions
  - Comples expressions reorganization
  - Excessive/redundant tests (`if` vs `case`)
- Conditional expressions within cycles
  - Cycle (induction variable) independent
  - Cycle (induction variable) dependent
    - Iteration independent
    - Dependence between iterations

Conditional expressions – example

```
DO I=1,K
 IF (N .EQ 0) THEN
    A(I)=A(I)+B(I)*C
 ELSE
    A(I)=0
 ENDIF
```

$\Longrightarrow$

```
IF (N .EQ 0) THEN
  DO I=1,K
    A(I)=A(I)+B(I)*C
  CONTINUE
ELSE
  DO I=1,K
    A(I)=0
  CONTINUE
ENDIF
```

Garbage elimination II

- Reduction
  - min (or max):
    ```
    for(i=0;i<n;i++)
       z=(a[i] > z) ?  a[i] :  z;
    ```
  - how to deal with a recursive dependency:
    ```
    for(i=0;i<n-1;i+=2) {
       z0=(a[i] > z0) ?  a[i] :  z0;
       z1=(a[i+1] > z1) ?  a[i+1] :  z1;
    }
    z=(z0 < z1) ?  z1 :  z0;
    ```

Reduction – Associative transformations

- Numerical imprecision:
  4 valid decimal digits

```
(X + Y) + Z = (.00005 + .00005) + 1.0000
               .00010 +         + 1.0000 = 1.0001
ale
X + (Y + Z) = .00005 + (.00005 + 1.0000) =
              .00005           + 1.0000 = 1.0000
```

- Reduction

```
DO I=1,N
   SUM=SUM+A(I)*B(I)
```

  Reduction with recursive dependency – can we use the same trick as with `min` reduction?

Garbage elimination III

- Branches (jumps)
- Type conversion

```
REAL*4 A(1000)
REAL*8 B(1000)
DO I=1,1000
  A(I)=A(I)*B(I)
```

- Manual optimization
  - Common subexpressions
  - Code move
  - Array processing (intelligent compiler, C and pointers)

Cycle optimization

- Goals:
  - Overhead reduction
  - Better access to memory (efficient use of caches)
  - Parallelism increase

RAW, WAR and WAW dependencies

- Named according how variables are used in the code (two occurencies)
- **Read after Read (RAR)**
  - "Benign" (in fact no) dependency
- **Read after Write (RAW)**
  - "True" dependency
  - Most problematic, order cannot be changed
- **Write after Read (WAR)**
  - "Antidependency"
  - Can be dealt with by *renaming*
- **Write after Write (WAW)**
  - "Output" dependency
  - Order cannot change unless checked for other dependencies

Data dependencies I

- **Flow Dependencies** (backward dependencies)
  - Example: `A(2:N) = A(1:N-1)+B(2:N)`

```
DO I=2,N                          DO I=2,N,2
  A(I)=A(I-1)+B(I)      ⟹          A(I)=A(I-1)+B(I)
                                    A(I+1)=A(I-1)+B(I)+B(I+1)
```

- **Anti-Dependencies**
  - Variable renaming as a default solution
  - Example:

```
                                  DO I=1,N
                                    A'(I)= B(I) * E
DO I=1,N                          DO I=1,N
  A(I) = B(I) * E        ⟹          B(I) = A(I+2) * C
  B(I) = A(I+2) * C               DO I=1,N
                                    A(I) = A'(I)
```

Data dependencies II

- **Output Dependencies**
  - Example:

    ```
    A(I)   = C(I) * 2
    A(I+2) = D(I) + E
    ```
  - Several values of a variable are computed during the cycle execution, but only the “last” is to be written
  - Not always easy to recognize, which value is “the last”

Loop unrolling I

- Cycle body copies several times within the cycle

```
DO I=1,N
  A(I)=A(I)+B(I)*C
```

$$\Downarrow$$

```
DO I=1,N,4
  A(I)=A(I)+B(I)*C
  A(I+1)=A(I+1)+B(I+1)*C
  A(I+2)=A(I+2)+B(I+2)*C
  A(I+3)=A(I+3)+B(I+3)*C
```

Loop unrolling II

- Major purpose
  - Overhead reduction
    - Reduction of number of iterations (=number of branches)
  - Parallelism increase (also within a single superscalar processor)
    - Software pipelining
- Pre- a postconditioning loops
  - Actual number of iterations adaptation

Loop unrolling III

- Unsuitable loops
  - Small number of iterations ⟶ full loop unrolling
  - “Fat” (=too large)) cycles: already include sufficient number of opportunities for parallelization
  - Loops with procedure calls: see also the procedure inlining
  - Loops with conditional expressions: more important for older processors
  - “Recursive” loops: with internal dependencies (cross iterations) (`a[i]=a[i]+a[i-1]*b`)

Loop unrolling problems

- Unrolling with a bad number of iterations
- Register clogging
- Misses of instruction cache (too long cycle)
- Hardware problems
    - esp. on multiprocessors with shared memory (cache coherency, bus overload, ...)
- Special cases: external loops unrolling, loops combination

Loops combination

- Repeated use of data (cache efficiency)
- Large loop body
- Compiler can do the combination if there is no code between loops

```
for(i=0;i<n;i++)
   a[i]=b[i]+1
for(i=0;i<n;i++)
   c[i]=a[i]/2
for(i=0;i<n;i++)
   d[i]=1/c[i+1]
```

$\Longrightarrow$

```
a[0]=b[0]+1
c[0]=a[0]/2
for(i=1;i<n;i++) {
   a[i]=b[i]+1
   c[i]=a[i]/2
   d[i-1]=1/c[i]
}
d[n]=1/c[n+1]
```

External loops unrolling

- Example:

```
DO I=1,N
   DO J=1,N
     A(I)=A(I)+B(I,J)*C(J)
```

- A(I) is a constant in the internal loop, C(J) is properly walked through
- B(I,J) Fortran has inverse order!

```
DO I=1,N,4
   DO J=1,N
     A(I+0)=A(I+0)+B(I+0,J)*C(J)
     A(I+1)=A(I+1)+B(I+1,J)*C(J)
     A(I+2)=A(I+2)+B(I+2,J)*C(J)
     A(I+3)=A(I+3)+B(I+3,J)*C(J)
```

Memory access optimization

- Optimal: smallest step (cache optimization)
- Work with arrays – C vs. Fortran
  - C: stored by rows, the right index is fastest to change
  - Fortran: stored by columns, the left index is fastest to change
- Index inversion
  - Example:

```
DO I=1,N
  DO J=1,N
    A(I,J)=B(I,J)+C(I)*D
```

$\Longrightarrow$

```
DO J=1,N
  DO I=1,N
    A(I,J)=B(I,J)+C(J)*D
```

Memory access optimization II

- Combination into blocks
  - Example:

```
DO I=1,N
  DO J=1,N
    A(J,I)=A(J,I)+B(I,J)
                   ⇓
DO I=1,N,2
  DO J=1,N,2
    A(J,I)=A(J,I)+B(I,J)
    A(J+1,I)=A(J+1,I)+B(I,J+1)
    A(J,I+1)=A(J,I+1)+B(I+1,J)
    A(J+1,I+1)=A(J+1,I+1)+B(I+1,J+1)
```

Cache optimization – Blocking

- A general technique to split loops working with arrays to loops that work on a block of arrays
- Example: Matrix transposition – `c` is transpose of `a`

```
DO I=1,M
  DO J=1,N
    c[j+i*n] = a[i+j*m]
```

Easy, but for any large `n` and `m` the cache overflow occurs

```
DO I=1,M,B
  DO J=1,N,B
    DO II=1,B
      DO JJ=1,B
        c[j+jj+(i+ii)*n] = a[i+ii+(i+jj)*m]
```

B is the block size, derived from the data cache size

Memory access optimization III

- Indirect addressing
  - Example:

    `b[i]=a[i+k]*c`, value of `k` unknown in the compile time

    `a[k[i]] += b[i]*c`
- Use of pointers
- Insufficient memory capacity
  - "Manual" processing
  - Virtual memory

More reading

- http://www.inf.ed.ac.uk/teaching/courses/copt/