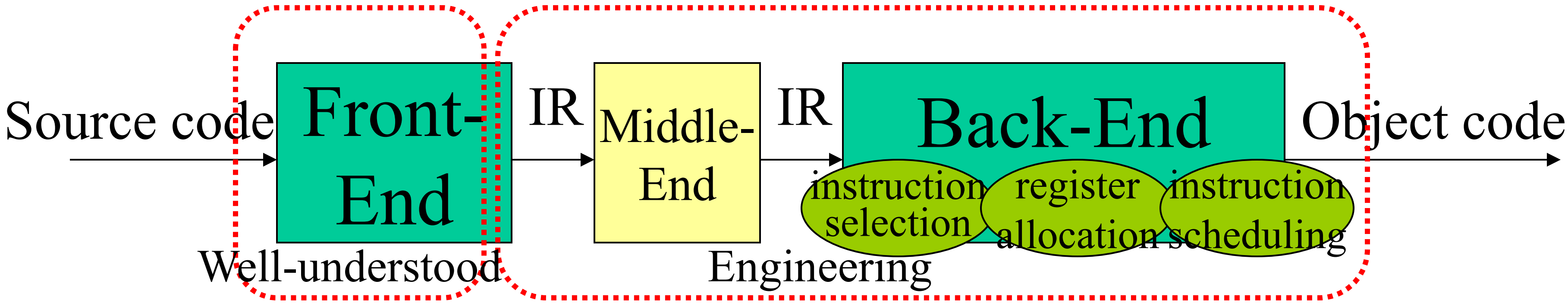
Lecture 19: Code Optimisation



- Code Optimisation: (could be a course on its own!)
  - Goal: improve program performance within some constraints.
    - (may also reduce size of the code, power consumption, etc…)
  - Issues:
    - Legality: must preserve the meaning of the program.
      - Externally observable meaning may be sufficient/may need flexibility.
    - Benefit: must improve performance on average or common cases.
      - Predicting program performance is often non-trivial.
    - Compile-time cost justified: list of possible optimisations is huge.
      - Interprocedural optimisations (O4).

Optimising Transformations

- Finding an appropriate sequence of transformations is a major challenge: modern optimisers are structured as a series of passes:
  - optimisation 1 is followed by optimisation 2; optimisation 2 is followed by optimisation 3, and so on…
- Transformations may improve program at:
  - Source level (algorithm specifics)
  - IR (machine-independent transformations)
  - target code (machine-dependent transformations)
- Some typical transformations:
  - Discover and propagate some constant value.
  - Remove unreachable/redundant computations.
  - Encode a computation in some particularly efficient form.

Classification

- By Scope:
  - Local: within a single basic block.
  - Peephole: on a window of instructions (usually local)
  - Loop-level: on one or more loops or loop nests.
  - Global: for an entire procedure
  - Interprocedural: across multiple procedures or whole program.
- By machine information used:
  - Machine-independent versus machine-dependent.
- By effect on program structure:
  - Algebraic transformations (e.g., x+0, x*1, 3*z*4, …)
  - Reordering transformations (change the order of 2 computations)
    - Loop transformations: loop-level reordering transformations.

Some transformations…

- Common subexpression elimination:
  - An expression, say x+y, is redundant iff along every path from the procedure's entry it has been evaluated and its constituent subexpressions (x, y) have not been redefined.
- Copy propagation:
  - After a 'copy' statement, x=y, try to use y as far as possible.
- Constant propagation:
  - Replace variables that have constant values with these values.
- Constant folding:
  - Deduce that a value is constant, and use the constant instead.
- Dead-code elimination:
  - A value is computed but never used; or, there is code in a branch never taken (may result after constant folding).
- Reduction in strength:
  - Replace x/4.0 with x*0.25

Examples

Before optimisation	After optimisation
`// Common subexpression elimination` `A[I,I*2+10]=B[I,I*2+10]+5`	`tmp=I*2+10` `A[I,tmp]=B[I,tmp]+5`

`// Copy propagation`
`t=I*4`
`s=t`
`a[s]=a[s]+4`
	`t=I*4`
	`s=t`
	`a[t]=a[t]+4`

`// Constant propagation`
`N=64`
`c=2`
`for (I=0;I<N;I++)`
`   a[I]=a[I]+c`
	`N=64`
	`c=2`
	`for (I=0;I<64;I++)`
	`   a[I]=a[I]+2`

`// Constant folding` `tmp=5*3+8-12/2`	`tmp=17`

`// Dead-code elimination`
`if (3>7) then { … }`
	`// removed (some of the`
	`// above optimisations may`
	`// create ` `` ` ``useless' code…)`

`// Reduction in strength` `x*2+x*1024`	`x+x+(x<<10)`

Loop Transformations

- **Loop-invariant code-motion:**
  - Detect statements inside a loop whose operands are constant or have all their definitions outside the loop - move out of the loop.
- **Loop interchange:**
  - Interchange the order of two loops in a loop nest (needs to check legality): useful to achieve unit stride access.
- **Strip mining:**
  - May improve cache usage when combined with loop interchange.

Example: Applying stip mining + loop interchange (**loop tiling**)

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

→

```
DOALL JJ=1,N,SJ
  DOALL II=1,N,SI
    DO J=JJ,MIN(JJ+SJ-1,N)
      DO K=1,N
        DO I=II,MIN(II+SI-1,N)
          A(I,J)=A(I,J)+B(I,K)*C(K,J)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Loop Transformations – Loop Unrolling

- Change: **`for(i=0;i<n;i++)`** to **`for(i=0;i<n-s+1;i+=s)`** and replicate the loop body **`s`** times (changing also **`i`** as needed to **`i+1`**, **`i+2`**, etc…). Will need an 'epilogue' if **`s`** does not divide **`n`**.
- Creates larger basic blocks and facilitates instruction scheduling.

<u>Example:</u>

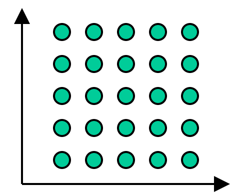
```
for (i=0; i<n; i++)  {
     a[i]=b[i]*c[i]; }
```

<u>After loop unrolling</u> it becomes:

```
for (i=0; i<n-s+1; i+=s) {
     a[i]=b[i]*c[i];
     a[i+1]=b[i+1]*c[i+1];
      … (the loop body repeated s times, from i to i+s-1)
}
/* epilogue */
for (j=i; j<n; j++) {
     a[j]=b[j]*c[j]; }
```

Is loop interchange legal? A more complex problem

- A model to represent loops: the polytope model.
  - Set of inequalities: $1 \leq i \leq 5$; $1 \leq j \leq 5$.
  - (Geometrical equivalence is possible)



- Locating data dependences between two statement instances in two different iterations of a loop (loop-carried dependence) is a complex problem.
  - What if the loop body contains a[i,j]=a[i-1,j-1]?
- A dependence vector is defined by the distance of two iterations that cause a dependence: for instance, [1,1] above.
- Complex analysis allows a formal framework to be developed.
- A loop nest of two loops can be interchanged when it has a dependence vector where both elements have the same sign.

Conclusion

- Program optimisation is a major research issue with several challenges: find an appropriate sequence of transformations (feedback-based, iterative compilation are amongst the ideas currently pursued); apply optimisations interprocedurally.
- Analysis for some transformations may be very expensive.
- Lots of work/research in several contexts...
- Reading: Aho2, Ch.9 (skim through Ch.11); Aho1 pp.585-602; Cooper, Ch.8 (for those interested further in the topic: Bacon et al, "Compiler Transformations for …", ACM Computing Surveys 26(4), 1994; M.Wolfe, High-Performance Compilers for Parallel Computing, Addison-Wesley; R.Allen & K.Kennedy, Optimizing Compilers for Modern Architectures, Morgan Kaufmann, 2002)