18-747 Lecture 15: Code Compilation for Superscalars

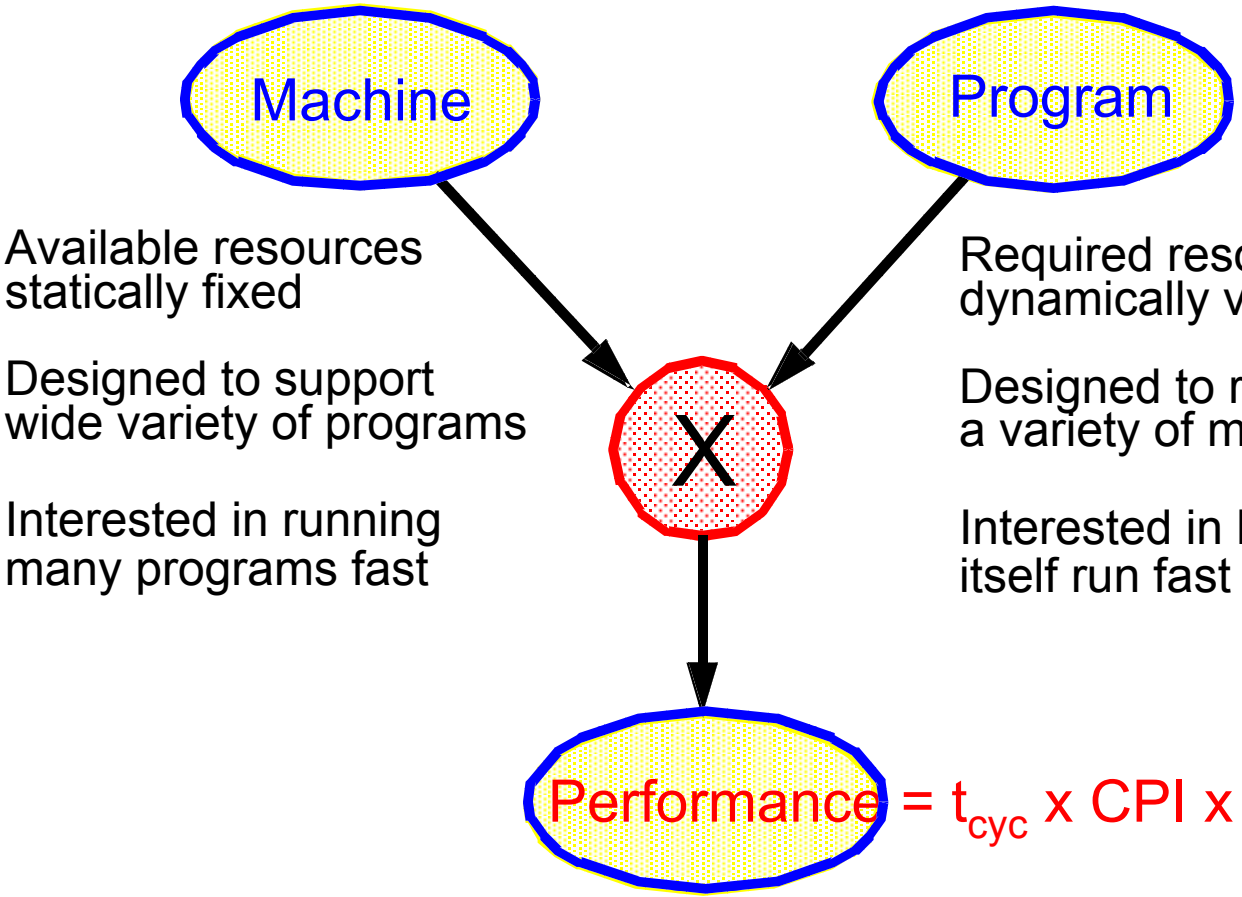
James C. Hoe
Dept of ECE, CMU
October 24, 2001

Reading Assignments: MJ Ch10

Announcements:

Handouts: *Handout 11: Revised Syllabus*
Handout 12: Homework #3
Graded Quiz 1
Quiz 1 Solutions

Hardware-Software Interface



Available resources statically fixed

Designed to support wide variety of programs

Interested in running many programs fast

Required resources dynamically varying

Designed to run well on a variety of machines

Interested in having itself run fast

Reflects how well the machine resources match the program requirements

Compiler Tasks

- Code Translation
  - Source language → target language

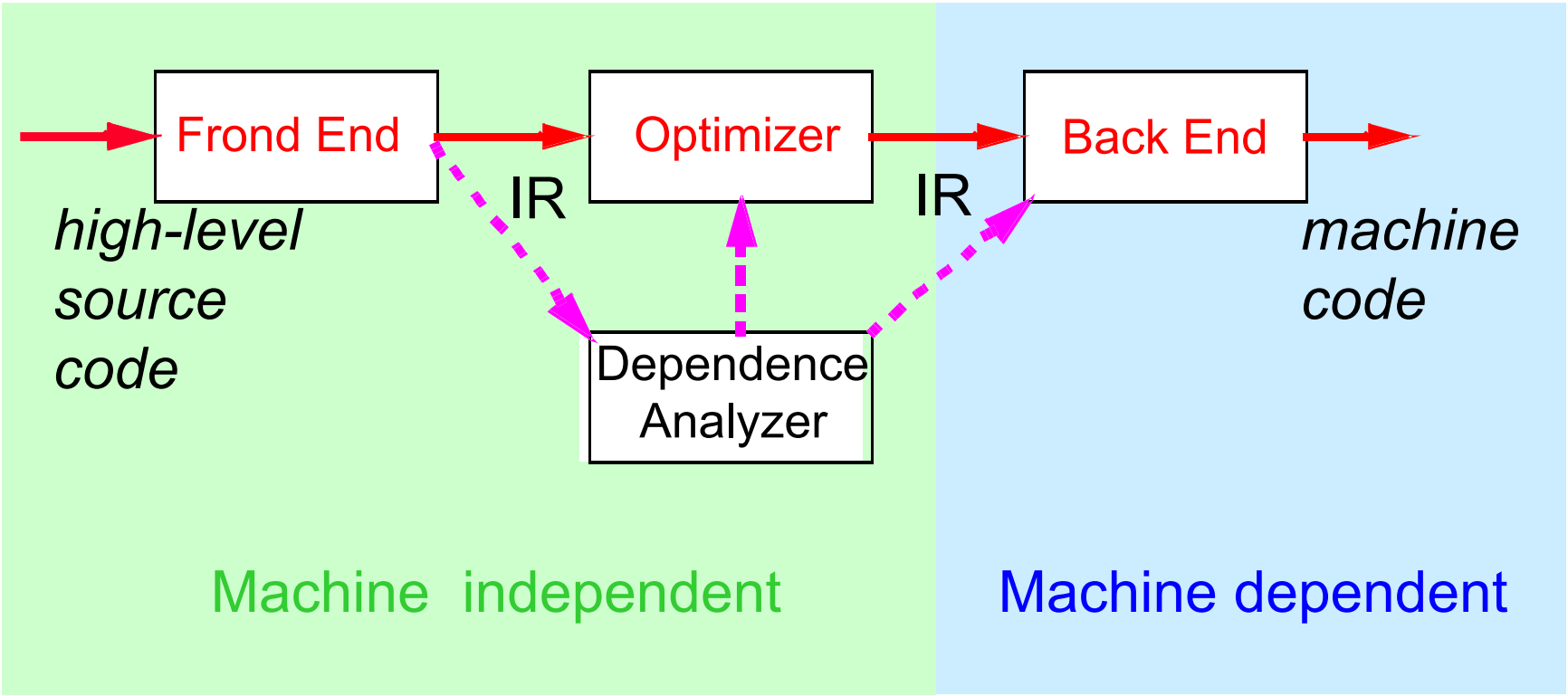
    FORTRAN → C

    C → MIPS, PowerPC or Alpha machine code

    MIPS binary → Alpha binary

- Code Optimization
  - Code runs faster
  - Match dynamic code behavior to static machine structure

Compiler Structure



(IR= intermediate representation)

Front End

- Lexical Analysis
  - Misspelling an identifier, keyword, or operator
    e.g. lex
- Syntax Analysis
  - Grammar errors, such as mismatched parentheses
    e.g. yacc
- Semantic Analysis
  - Type checking

Intermediate Representation

- Achieve retargetability
  - Different source languages
  - Different target machines
- Example (tree-based IR from CMCC)

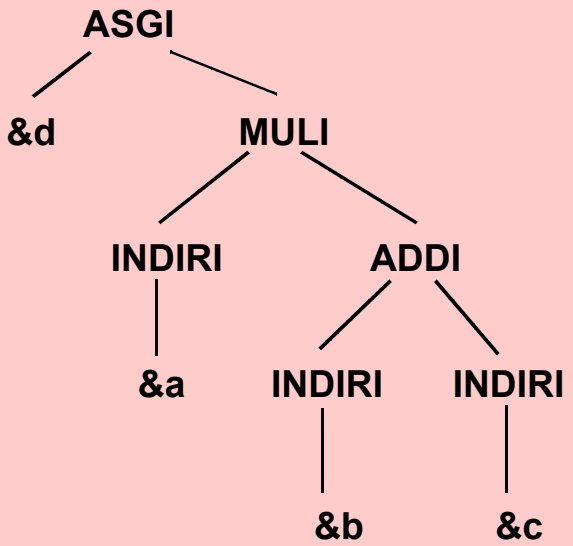
int a, b, c, d;
d = a * (b+c)

Linear form of graphical representation

```
A0   5   78   "a"
A1   5   78   "b"
A2   5   78   "c"
A3   5   78   "d"

FND1   ADDRL   A3
FND2   ADDRL   A0
FND3   INDIRI  FND2
FND4   ADDRL   A1
FND5   INDIRI  FND4
FND6   ADDRL   A2
FND7   INDIRI  FND6
FND8   ADDI    FND5   FND7
FND9   MULI    FND3   FND8
FND10  ASGI    FND1   FND9
```

Graphical Representation



Code-Optimizing Transformations

- Constant folding

  (1 + 2) ⇒ 3

  (100 > 0) ⇒ true

- Copy propagation

  ```
  x = b + c          x = b + c
  z = y * x    ⇒     z = y * (b + c)
  ```

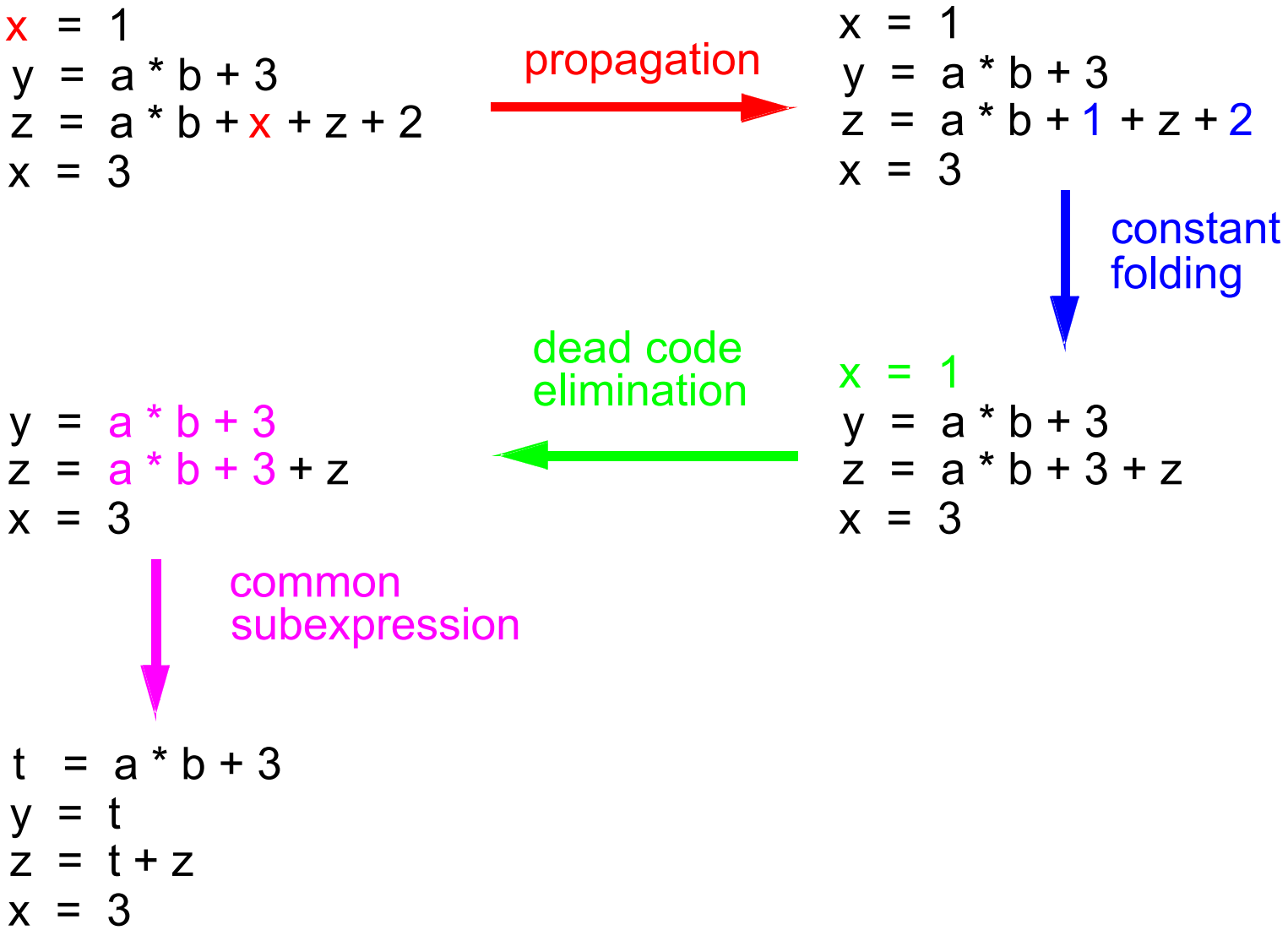
- Common subexpression

  ```
  x = b * c + 4      t = b * c
  z = b * c - 1  ⇒   x = t + 4
                     z = t - 1
  ```

- Dead code elimination

  ```
  x = 1
  x = b + c          or if x is not referred to at all
  ```

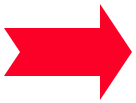
Code Optimization Example



Code Motion

- Move code between basic blocks
- E.g. move loop invariant computations outside of loops

```
while ( i < 100 ) {
        *p  =  x / y + i
        i  =  i + 1
}
```

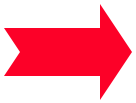


```
t  =  x / y
while ( i < 100 )  {
        *p  =  t + i
        i  =  i  + 1
}
```

Strength Reduction

- Replace complex (and costly) expressions with simpler ones
- E.g.

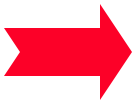
```
a : = b*17
```



```
a: = (b<<4) + b
```

- E.g.

```
while ( i < 100 ) {
        a[ i ] = i * 100
        i = i + 1
}
```



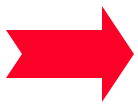
```
p = & a[ i ]
t = i * 100
while ( i < 100 ) {
        *p = t
        t = t + 100
        p = p + 4
        i = i + 1
}
```

loop invariant: &a[i]==p, i*100==t

Induction Variable Elimination

- Induction variables may become redundant
- Test replacement instead

```
p = & a[ i ]
t = i * 100
while ( i < 100 ) {
    *p = t
     t = t + 100
     p = p + 4
     i = i + 1
}
```



```
p = & a[ i ]
limit = p + 400
t = i * 100
while ( p < limit ) {
    *p = t
     t = t + 100
     p = p + 4
     i = i + 1
}
i = 100
```

Importance of Loop Optimizations

Program	No. of Loops	Static B.B. Count	Dynamic B.B. Count	% of Total
nasa7	9	---	322M	64%
	16	---	362M	72%
	83	---	500M	~100%
matrix300	1	17	217.6M	98%
	15	96	221.2M	98+%
tomcatv	1	7	26.1M	50%
	5	22	52.4M	99+%
	12	96	54.2M	~100%

Study of loop-intensive benchmarks in the SPEC92 suite *[C.J. Newburn, 1991]*

Loop Unrolling

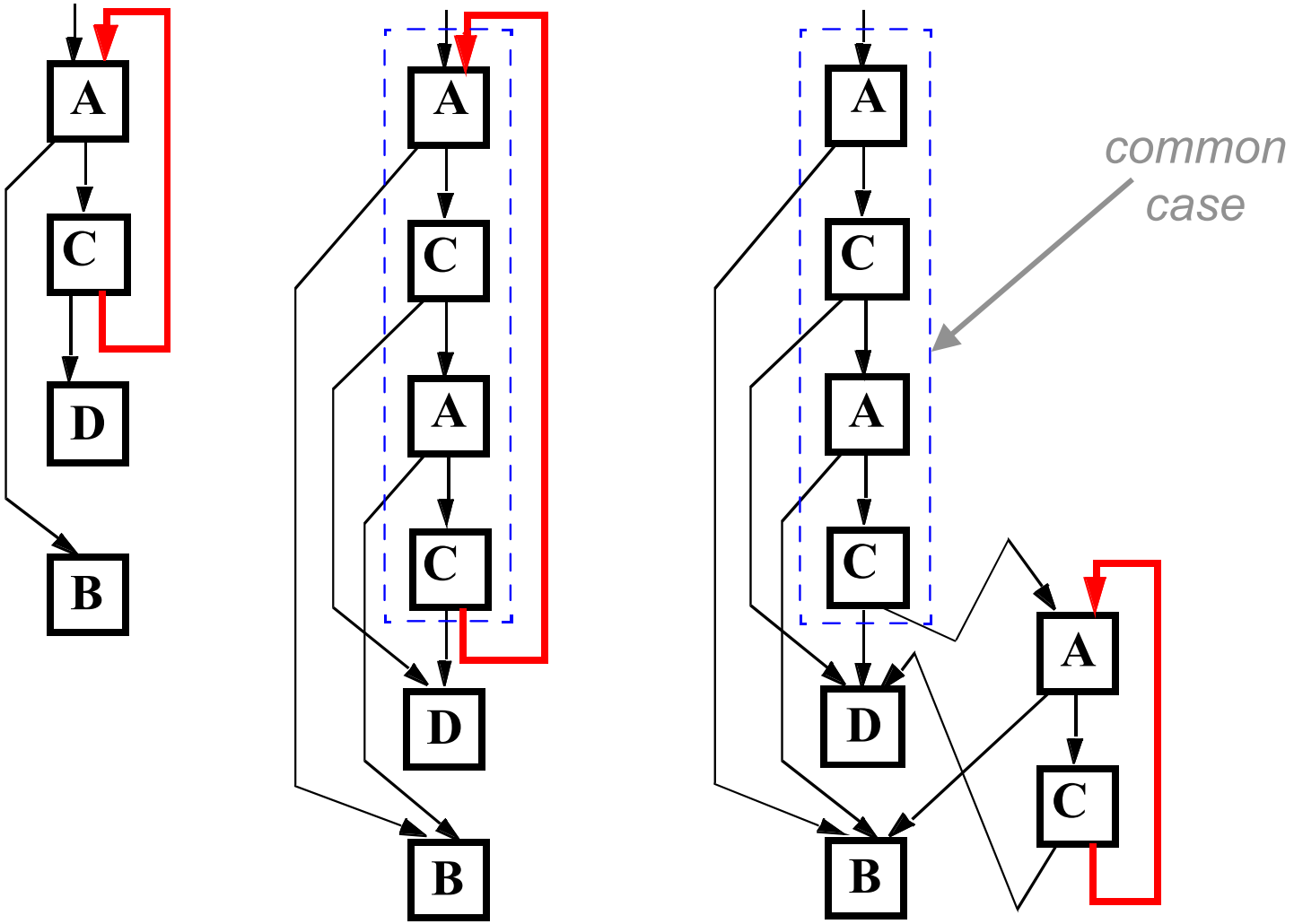
```
i = 1;
while ( i < 100 ) {
      a[i] = b[i+1] + (i+1)/m
      b[i] = a[i-1] - i/m
      i = i + 1
}
```

```
i = 1;
while ( i < 100 ) {
      a[i] = b[i+1] + (i+1)/m
      b[i] = a[i-1] - i/m

      a[i+1] = b[i+2] + (i+2)/m
      b[i+1] = a[i] - (i+1)/m
      i = i + 2
}
```

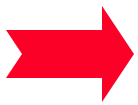
- Reduce loop overhead
  - increment induction var
  - loop condition test
- Enlarged basic block (and analysis scope)
  - Instruction-level parallelism
  - More common subexpression
  - Memory accesses (aggressive memory aliasing analysis)

Loop Unrolling vs. Loop Peeling



Software Pipelining

```
i=0
while (i<99) {
        ;; a[ i ]=a[ i ]/10
        x = a[ i ]
        y = x  / 10
        a[ i ] = y
        i++
}
```

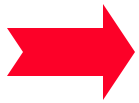


```
i=0
while (i<99) {
        x = a[ i ]
        y = x  / 10
        a[ i ] = y

        x = a[ i+1 ]
        y = x  / 10
        a[ i+1 ] = y

        x = a[ i+2 ]
        y = x  / 10
        a[ i+2 ] = y

        i=i+3
}
```



```
i=0
y=a[ 0 ] / 10
x=a[ 1 ]

while (i<97) {
        a[i]=y

        y=x / 10

        x=a[i+2]

        i++
}

a[97]=y
a[98]=x / 10
```

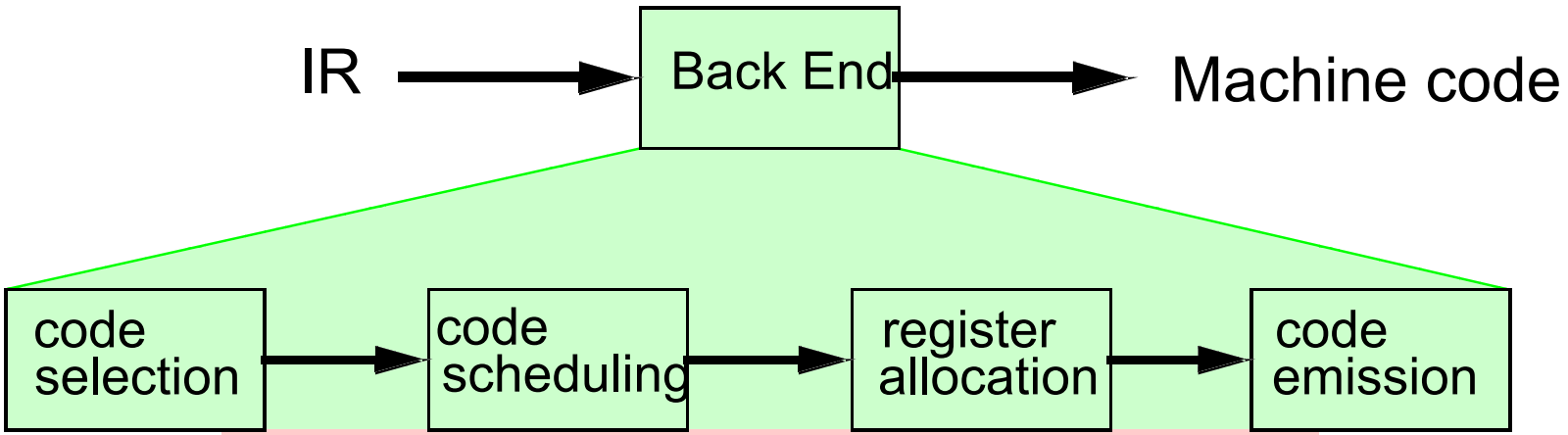
Function inlining

- Replace function calls with function body
- Increase compilation scope (increase ILP)
  e.g. constant propagation, common subexpression
- Reduce function call overhead
  e.g. passing arguments, reg. saves and restores

[W.M. Hwu, 1991 (DEC 3100)]

Program	In-line Speedup	in-line Code Expansion
cccp	1.06	1.25
compress	1.05	1.00+
equ	1.12	1.21
espresso	1.07	1.09
lex	1.02	1.06
tbl	1.04	1.18
xlisp	1.46	1.32
yacc	1.03	1.17

Back End



- map virtual registers into architect registers
- rearrange code
- target machine specific optimizations
  - delayed branch
  - conditional move
  - instruction combining
    - auto increment addressing mode
    - add carrying (PowerPC)
    - hardware branch (PowerPC)

Instruction-level IR