תורת הקומפילציה
הרצאה 13

אופטימיזציה

מה זה אופטימיזציה?

טרנספורמציה של תכנית המקומפלת **לצורך שיפור** ביצועי התוכנית

מגוון קריטריונים:

- אופטימיזציה של זמן ריצה (נפוצה ביותר)
- אופטימיזציה של גודל הקוד
- אופטימיזציה של צריכת זיכרון
- אופטימיזציה של צריכת חשמל

אופטימיזציה - אתגרים ומגבלות

- **חייבת לשמור על הסמנטיקה** של התכנית – אופטימיזציה בטוחה (**safe**)
  - התוכנית המתקבלת צריכה להיות **שקולה** לתוכנית המקורית
  - דורש ניתוח לא טריוויאלי של התוכנית

- בניגוד למה שמשתמע מהשם, **בדרך כלל לא מגיעים לאופטימום**

- **לא משפרת אלגוריתם** שאינו יעיל

- **לא מתקנת באגים**

<u>דילמה קלאסית</u>

- כמה זמן שווה להשקיע בזמן קומפילציה כדי לחסוך בזמן ריצה?
  תלוי בדרישות; למשל – אופטימיזציה של גודל קוד וצריכת זיכרון :
  מאוד חשובה עבור שבב עם זיכרון קטן, פחות מעניינים עבור תחנת עבודה.

למה יש חוסר יעילות בתוכנית?

- יתירות (redundancy) בתוכנית המקור:
  - נוצרות בעקבות האצה של כתיבת הקוד על ידי המתכנת (copy-paste);
    למשל, חישוב של ביטוי פעמיים
  - תורמות לקריאות של הקוד

  הנחת המתכנת: הקומפיילר יסלק את היתירות.

- יתירות כתוצאה מכתיבה בשפה עילית:

  למשל, בפקודה a[i] = a[i]+1 הפנייה למערך a[i] מתרגמת ל- a+4*i. כתוצאה:
  - אותו חישוב מתבצע פעמיים
  - וזה יכול לחזור על עצמו בתוך <u>לולאה</u>

- יתירות כתוצאה מהתרגום:
  – כיון שהתרגום מתבצע באופן אוטומטי ו"לוקלי", לא תמיד יוצא קוד חכם.

Optimization process - big picture

- Each local optimization does little by itself
  - Some look non-significant, but together are very efficient
- Typically optimizations interact
  - Performing one optimization enables another
- Optimization passes are repeated over and over, until no improvement is possible
  - can be controlled to limit compilation time ;

    e.g. – by selecting one of the predefined levels of optimization
- Ordering of optimizations is often arbitrary

Constant Folding

Idea

If operands are known at compile-time, evaluate expression at compile-time

Constant Folding

Idea

If operands are known at compile-time, evaluate expression at compile-time

Various reasons for this redundancy:

- appears in the source program

```
r = 3.141 * 10;
```

⇒

```
r = 31.41;
```

Constant Folding

Idea

If operands are known at compile-time, evaluate expression at compile-time

Various reasons for this redundancy:

- appears in the source program

```
r = 3.141 * 10;
```

⇒

```
r = 31.41;
```

- result of translation to intermediate code:

```
x = A[2];
```

⇒

```
t1 = 2*4;
t2 = A + t1;
x = *t2;
```

⇒

```
t1 = 8;
t2 = A + t1;
x = *t2;
```

Constant Folding

Idea

If operands are known at compile-time, evaluate expression at compile-time

Various reasons for this redundancy:

- appears in the source program

```
r = 3.141 * 10;
```

➡

```
r = 31.41;
```

- result of translation to intermediate code:

```
x = A[2];
```

➡

```
t1 = 2*4;
t2 = A + t1;
x = *t2;
```

➡

```
t1 = 8;
t2 = A + t1;
x = *t2;
```

- result of other optimizations

  (e.g. constant propagation - see later)

Algebraic Simplification

Idea:
- Apply algebraic rules to simplify expressions

`a * 1`	⇒	`a`
`a/1`		`a`
`a * 0`		`0`
`a + 0`		`a`
`a || false`		`a`
`a && true`		`a`

Algebraic Simplification

Idea:
- Apply algebraic rules to simplify expressions

`a * 1`	⇒	`a`
`a/1`		`a`
`a * 0`		`0`
`a + 0`		`a`
`a || false`		`a`
`a && true`		`a`

Safety: no side effects in *a* (e.g. when *a* is a function call)

Algebraic Simplification

Idea

- Apply algebraic rules to simplify expressions

`a * 1`	`a`
`a/1`	`a`
`a * 0`	`0`
`a + 0`	`a`
`a || false`	`a`
`a && true`	`a`

Safety: no side effects in *a* (e.g. when *a* is a function call)

Use associativity and commutativity rules;

combine with constant folding:

(2+a) + 4 → (a+2) + 4 → a + (2+4) → a+6

Constant Propagation

Idea

- If the value of a variable is known at compile-time, replace the use of variable with its value

```
int x = 5;
int y = x * 2;
int z = a[y];
```

```
int x = 5;
int y = 5 * 2;
int z = a[y];
```

- Value of the variable is propagated <u>forward</u> from the point of assignment

Constant Propagation

Idea

- If the value of a variable is known at compile-time, replace the use of variable with its value

```
int x = 5;
int y = x * 2;
int z = a[y];
```

```
int x = 5;
int y = 5 * 2;
int z = a[y];
```

- Value of the variable is propagated forward from the point of assignment

Safety Value is not changed between the point where it is set and the point of use

Constant Propagation

Idea

- If the value of a variable is known at compile-time, replace the use of variable with its value

```
int x = 5;
int y = x * 2;
int z = a[y];
```

```
int x = 5;
int y = 5 * 2;
int z = a[y];
```

```
int x = 5;
int y = 10;
int z = a[y];
```

- Value of the variable is propagated forward from the point of assignment

Safety Value is not changed between the point where it is set and the point of use

Combine with constant folding !

Constant Propagation

Idea

- If the value of a variable is known at compile-time, replace the use of variable with its value

```
int x = 5;
int y = x * 2;
int z = a[y];
```

```
int x = 5;
int y = 5 * 2;
int z = a[y];
```

```
int x = 5;
int y = 10;
int z = a[y];
```

- Value of the variable is propagated forward from the point of assignment

Safety Value is not changed between the point where it is set and the point of use

Combine with constant folding !

Repeat again

```
int x = 5;
int y = 10;
int z = a[10];
```

Copy Propagation

Idea

- After an assignment x = y, replace uses of x with y

```
x = y;
if (x>1)
  s = x+f(x);
```

⇒

```
x = y;
if (y>1)
  s = y+f(y);
```

Copy Propagation

Idea

- After an assignment x = y, replace uses of x with y

```
x = y;
if (x>1)
  s = x+f(x);
```

```
x = y;
if (y>1)
  s = y+f(y);
```

Copying may appear in the source code, and …

Often occurs after translation to intermediate code

```
x = a + b + c;
y = x + 1;
```

```
t1 = a + b;
t2 = t1 + c;
x = t2;
t3 = x + 1;
y = t3
```

```
t1 = a + b;
t2 = t1 + c;
x = t2;
t3 = t2 + 1;
y = t3
```

Copy Propagation

Idea

- After an assignment x = y, replace uses of x with y

```
x = y;
if (x>1)
  s = x+f(x);
```

```
x = y;
if (y>1)
  s = y+f(y);
```

Safety

- Only apply up to another assignment to x, **or**
- …another assignment to y!

```
x = y;
if (x>1)
  s = x+f(x);
x = t+1;
z = x*2;
```

don't replace x by y here !

```
x = y;
if (x>1)
  s = x+f(x);
y = w+3;
z = x-4;
```

Common Sub-Expression Elimination

Idea

- If program computes the same expression multiple times, reuse the value.

```
a = b + c;
c = b + c;
d = b + c;
```

```
t = b + c
a = t;
c = t;
d = b + c;
```

Common Sub-Expression Elimination

Idea

- If program computes the same expression multiple times, reuse the value.

```
a = b + c;
c = b + c;
d = b + c;
```

```
t = b + c
a = t;
c = t;
d = b + c;
```

can't replace b+c by t here!

Safety

Can reuse a subexpression until its operands are redefined

Common Sub-Expression Elimination

Idea

- If program computes the same expression multiple times, reuse the value.

```
a = b + c;
c = b + c;
d = b + c;
```

⇒

```
t = b + c
a = t;
c = t;
d = b + c;
```

Often occurs in address computations

- Array indexing and struct/field accesses

  a[i,j] = a[i,j] *2 ;

  s.f = s.f + 10;

- In source code - such computations are hidden
- In intermediate code – become explicit

Dead Code Elimination

Idea: If the result of a computation is never used, then we can remove the computation

dead code

```
x = y + 1;
y = 1;
x = 2 * z;
```

```
y = 1;
x = 2 * z;
```

Dead Code Elimination

Idea: If the result of a computation is never used, then we can remove the computation

dead code

```
x = y + 1;
y = 1;
x = 2 * z;
```

```
y = 1;
x = 2 * z;
```

Safety

- Variable is redefined on the way to all its uses
- No side effects in removed code
- In general, requires non-trivial analysis of all execution paths starting at the candidate command:

```
x = y + 1;
y = 1;
if a > 0 then x = 2 * z else a = 7 ;
t = x - 10;
```

- **this code is not dead !**
- **value of x is not redefined on this path to the use of x**

Dead Code Elimination

Idea: If the result of a computation is never used, then we can remove the computation

dead code

```
x = y + 1;
y = 1;
x = 2 * z;
```

⇒

```
y = 1;
x = 2 * z;
```

Often occurs after other optimizations

(for example - after constant/copy propagation):

```
pi = 3.14 ;
s = pi*r*r ;
```

⇒

```
pi = 3.14 ;
s = 3.14*r*r ;
```

dead code

```
x = y;
if (x>1)
   s = x+f(x);
```

⇒

```
x = y;
if (y>1)
   s = y+f(y);
```

dead code

Unreachable Code Elimination

Idea

- Eliminate code that can never be executed:
  - functions that are never called
  - branches that are never traversed

```
x = 10;
. . .
if (x < 0) y = 10 else y = 100;
```

```
x = 10;
. . .
y = 100;
```

Removing unreachable code

- makes the program smaller
- sometimes also faster, due to memory cache effects

Safety

Traverse program’s control flow graph, and mark the reachable linear blocks

Unreachable Code Elimination

<u>Often occurs</u> after other optimizations:

```
x = 10;
. . .
if (x < 0) y = 10 else y = 100;
```

constant propagation

```
x = 10;
. . .
if (10 < 0) y = 10 else y = 100;
```

constant folding

```
x = 10;
. . .
if (false) y = 10 else y = 100;
```

```
x = 10;
. . .
y = 100;
```

Unreachable Code Elimination

Sometimes is based on “symbolic” computations:

```
x = y + 2;
. . .
if (y > x) y = 10 else y = 100;
```

```
x = 10;
. . .
y = 100;
```

```
x = y * y;
. . .
if (x < 0) y = 10 else y = 100;
```

```
x = 10;
. . .
y = 100;
```

Requires a different type of analysis

Loop Optimizations

Program hot-spots are usually in loops

- Most programs: 90% of execution time is in loops

Loops are a good place to expend extra effort

- Numerous loop optimizations
- Very effective
- Many are more expensive optimizations

Loop-Invariant Code Motion

Idea

- If a computation won't change from one loop iteration to the next, move it outside the loop

```
for (i=0;i<N;i++)
  A[i] = A[i] + x*x;
```

⇒

```
t1 = x*x;
for (i=0;i<N;i++)
  A[i] = A[i] + t1;
```

Safety

- Determine when arguments of expression are invariant (don't change their values in the loop)
- Again – analysis of program's control flow graph

Strength Reduction – machine independent

Idea:

- Replace expensive operations (mult, div) with cheaper ones (add, sub)

Traditionally applied to induction variables

- Variables whose value depends linearly on loop count
- Special analysis to find such variables

```
for (i=0;i<N;i++)
  v = 4*i;
  A[v] = z
```

⇒

```
v = 0;
for (i=0;i<N;i++)
  A[v] = z
  v = v + 4;
```

NOTE: values of variable v : 0, 4, 8, …

Strength Reduction – machine oriented

Idea:

- Replace expensive operations (mult, div) with cheaper ones available in the target language

$x := x * 8 \Rightarrow x := x \ll 3$

$x := x * 15 \Rightarrow t := x \ll 4;\ x := t - x$

$a / 32767 \rightarrow (a >> 15) + (a >> 30)$

Loop unrolling

<u>Idea:</u> replace the body of a loop by several copies of the body and adjust the loop-control code

```
for (int i=0; i<100; i=i+1)
  { s = s + a[i]; }
```

```
for (int i=0; i<99; i=i+2)
  { s = s + a[i];
    s = s + a[i+1]; }
```

<u>Trade-off</u>

- Reduces the overhead of branching and checking the loop control
- Yields larger code (might impact the instruction cache)

<u>Which loops to unroll and by what factor?</u>

- Use heuristics; e.g.: loop body is a straight-line code

<u>Improves effectiveness of other optimizations</u>

(e.g. – elimination of common sub-expressions)

Inlining

<u>Idea</u>

Replace **a <u>non-recursive</u> function** call with the adjusted (according to the call's parameters) body of the function

<u>Safety</u>

Based on analysis of the function call graph

<u>Risk</u>

- Code size
- Most compilers use heuristics to decide when

<u>Critical</u> for OO languages

- Methods are often small

Example

- Initial code:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

- Strength Reduction

```
– a := x ** 2          – a := x ^ x
  b := 3                 b := 3
  c := x                 c := x
  d := c * c             d := c * c
– e := b * 2           – e := b << 1
  f := a + d             f := a + d
  g := e * f             g := e * f
```

- Copy propagation:

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

- Copy propagation:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

- Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

- Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

- Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

- Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

- Copy propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

- Copy propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

- Dead code elimination:

a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f

- Dead code elimination:

a := x * x

f := a + a
g := 6 * f

- This is the final form