Compiler Design

Fatemeh Deldar

Isfahan University of Technology

1403-1404

Code Optimization

Code Optimization

- ***Elimination of unnecessary instructions in object code*, or *the replacement of one sequence of instructions by a faster sequence of instructions that does the same thing* is usually called "code improvement" or "code optimization“**
- **Causes of Redundancy**
  - Sometimes the redundancy is available at the **source level**
    - For instance, a programmer may find it more direct and convenient to recalculate some result
  - But more often, the redundancy is a side effect of **having written the program in a high-level language**

در مرحله بهینه‌سازی تلاش می‌شود روی قسمت‌هایی از برنامه بهینه‌سازی صورت گیرد که تأثیر زیادی روی بالا بردن سرعت اجرای برنامه نهایی و یا کاهش حجم کد برنامه نهایی داشته باشد.

Common Optimization Techniques

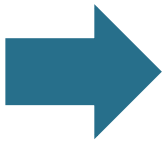
1. حذف زیرعبارت مشترک (Common subexpression elimination)

```
a = b + c
d = b + c
e = a + d * e
```

b + c یک زیر عبارت مشترک است، می‌توان یک بار آن را محاسبه کرد.

2. انتشار کپی (Copy propagation)

```
x = y;
if (x > 1) {
x = x * f(x – 1);
}
```



```
x = y;
if (y > 1) {
x = y * f(y – 1);
}
```

- اگر در دنباله‌ای از دستورات متوالی دستوری به شکل a=b (دستور کپی) داشته باشیم، تا جایی که a و b تغییر نکرده باشند، می‌توان به جای a از b استفاده کرد.
- این تبدیل به خودی خود برنامه را بهبود نمی‌دهد ولی زمینه را برای اعمال سایر تکنیک‌های بهینه‌سازی به وجود می‌آورد.

Common Optimization Techniques

3. حذف کد مرده (Dead code elimination)

کد مرده به دستوراتی گفته می‌شود که اجرا می‌شوند ولی اجرای آن‌ها تأثیری در منطق برنامه ندارد.

4. حذف کد غیر قابل دسترسی (Unreachable code elimination)

کد غیر قابل دسترسی کدی است که برای اجرا کنترل به آن داده نمی‌شود. به‌عنوان مثال، اگر در یک عبارت شرطی if-else در زمان کامپایل بدانیم که شرط همیشه درست است، بخش else هیچ وقت اجرا نمی‌شود و می‌توان آن را حذف کرد.

5. جایگذاری ثابت (Constant folding)

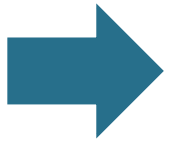
*int x = (2+3) * y* ➔ *int x = 5 * y*

اگر مقدار عملوند در زمان کامپایل مشخص است، مقدار عبارت را می‌توان محاسبه کرد.

Common Optimization Techniques

6. باز کردن حلقه (Loop unrolling)

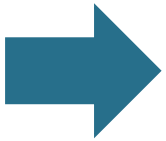
```
for (i=1; i<3; i++) {
  A[i] = j + i;
  B[i] = B[j – i];
}
```



```
A[1] = j + 1;
B[1] = B[j – 1];
A[2] = j + 2;
B[2] = B[j – 2];
```

- با تکرار دستورات بدنه حلقه به تعداد مورد نیاز، حلقه را حذف می‌کند.
- دستورات پرش را حذف می‌کند و در نتیجه موجب افزایش سرعت اجرا می‌شود، ولی معمولاً حجم کد را افزایش می‌دهد.

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



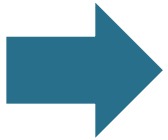
```
for (int i=0; i<99; i=i+2) {
  s = s + a[i];
  s = s + a[i+1];
}
```

Common Optimization Techniques

7. جانشینی رو به عقب (Back-substitution)

بدنه تابع جایگزین دستور فراخوانی تابع می‌شود و به این ترتیب سربار فراخوانی تابع از بین می‌رود و باعث افزایش سرعت برنامه می‌شود، ولی معمولاً حجم کد را افزایش می‌دهد.

```
int g(int x) { return x + pow(x); }
int pow(int a) {
  int b = 1; int n = 0;
  while (n < a) { b = 2 * b };
  return b;
}
```

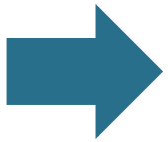


```
int g(int x) {
  int a = x;
  int b = 1; int n = 0;
  while (n < a) {b = 2 * b};
  tmp = b;
  return x + tmp;
}
```

Common Optimization Techniques

8. کشیدن به چپ و راست (Left and right hoisting)

```
if (a<b) {
  i = 1;
  …
  flag = i;
}
else {
  i = 1;
  …
  flag = i;
}
```



```
i = 1;
if (a<b) {
  …
}
else {
  …
}
flag = i;
```

- در ساختارهای شرطی if-else و switch-case کاربرد دارد.
- اگر کد مشترکی در ابتدا یا انتهای بخش‌های if و else وجود داشته باشد می‌توان آن را به قبل یا بعد از عبارت شرطی منتقل کرد.
- این تبدیل کاهش حجم کد را به همراه دارد ولی روی افزایش سرعت تأثیری ندارد.

Control Flow Graph

- بلوک پایه (Basic block)
  - هر بلوک پایه از تعدادی دستور تشکیل شده است که به‌صورت ترتیبی اجرا می‌شوند
  - آخرین دستورالعمل یک بلوک پایه می‌تواند دستور پرش باشد
- گراف جریان کنترل (Control flow graph)
  - بلوک‌های پایه گره‌های گراف جریان کنترل هستند و یال‌های این گراف مشخص می‌کنند که کنترل از چه بلوکی به چه بلوک دیگر می‌تواند برود
- ایجاد گراف جریان کنترل
  - هر دستورالعملی که مقصد پرش است شروع یک بلوک پایه است
  - هر دستورالعملی که بلافاصله بعد از یک پرش است شروع یک بلوک پایه است

Control Flow Graph

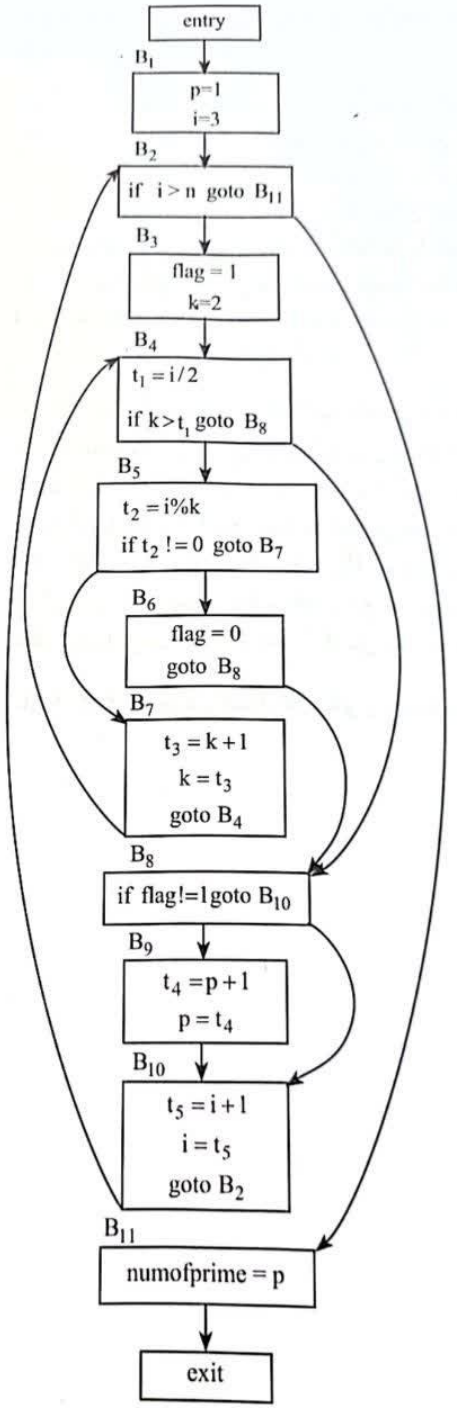
- **مثال:** بلوک‌های پایه و گراف جریان کنترل را در کد روبرو به دست آورید.

تعداد اعداد اول بین 1 تا n را محاسبه می‌کند.

```
1:   p = 1
2:   i = 3
3:   if i>n goto 21
4:   flag = 1
5:   k = 2
6:   t1 = i / 2
7:   if k>t1 goto 15
8:   t2 = i % k
9:   if t2 != 0 goto 12
10:  flag = 0
11:  goto 15
12:  t3 = k + 1
13:  k = t3
14:  goto 6
15:  if flag != 1 goto 18
16:  t4 = p + 1
17:  p = t4
18:  t5 = i + 1
19:  i = t5
20:  goto 3
21:  numofprime = p
```

Control Flow Graph

- **مثال (ادامه): بلوک‌های پایه و گراف جریان کنترل را در کد روبرو به دست آورید.**



Control Flow Graph

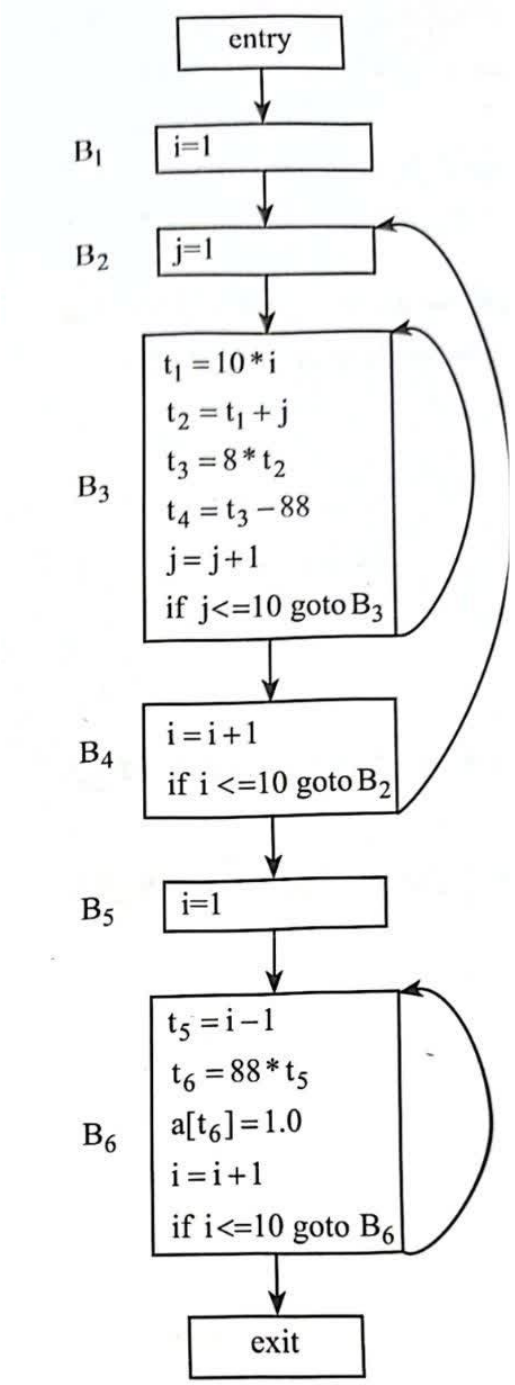
- **مثال:** گراف جریان کنترل را در کد میانی روبرو به دست آورید.

یک ماتریس ۱۰ در ۱۰ را به یک ماتریس همانی تبدیل می‌کند.

```
1)    i = 1
2)    j = 1
3)    t1 = 10 * i
4)    t2 = t1 + j
5)    t3 = 8 * t2
6)    t4 = t3 - 88
7)    A [t4] = 0.0
8)    j = j + 1
9)    if  j <= 10  goto (3)
10)   i = i + 1
11)   if  i <= 10  goto (2)
12)   i = 1
13)   t5 = i – 1
14)   t6 = 88 * t5
15)   A [t6] = 1.0
16)   i = i + 1
17    if  i <= 10  goto (13)
```

Control Flow Graph

- **مثال (ادامه): گراف جریان کنترل را در کد میانی روبرو به دست آورید.**



Code Optimization

- **Local code optimization**
  - Code improvement within a basic block
- **Global code optimization**
  - Code improvement across basic blocks

Local Optimization

- **Many important techniques for local optimization begin by transforming a basic block into a DAG (Directed Acyclic Graph)**

- ساخت DAG

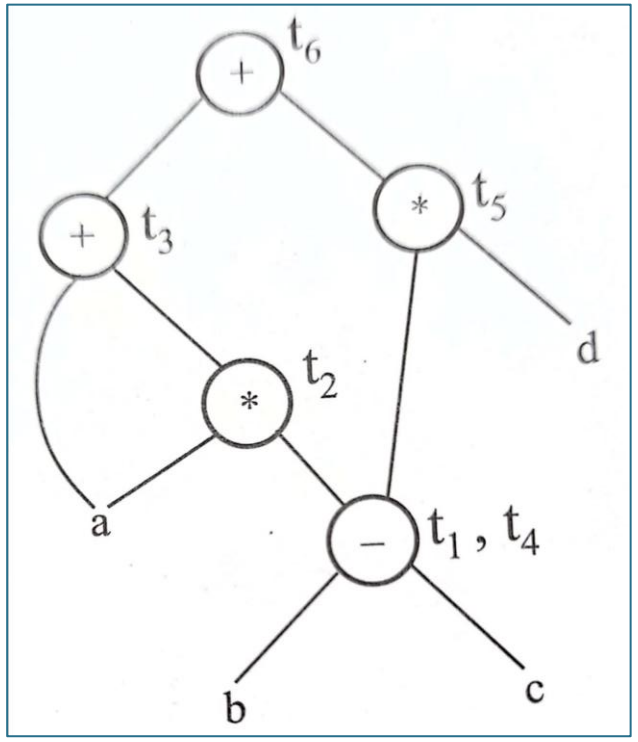
1. برای هر یک از مقادیر اولیه متغیرهای ظاهرشده در بلوک پایه یک گره در نظر بگیرید.
2. متناظر با هر دستور S در بلوک پایه یک گره N در گراف وجود دارد. فرزندان N گره‌های متناظر با دستوراتی هستند که آخرین مقداردهی عملوندهای استفاده شده در S توسط این دستورات انجام شده است.
3. گره N توسط عملگر به‌کار گرفته شده در دستور S برچسب‌گذاری می‌شود. همچنین به گره N لیستی از متغیرهایی که این گره برای آن‌ها آخرین مقداردهی است منسوب می‌شود.

گره‌های خروجی (Output nodes) گره‌هایی هستند که مقادیر آن‌ها ممکن است بعداً در بلاک‌های دیگر گراف استفاده شود. به این گره‌ها Live on exit می‌گویند.

Local Optimization

- **Example:** **Convert basic block to GAG**

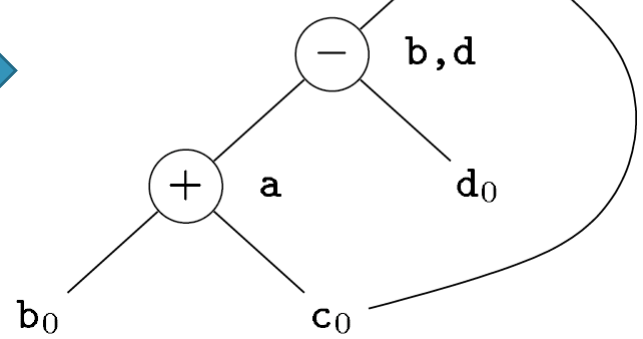
```
t1 = b – c
t2 = a * t1
t3 = a + t2
t4 = b – c
t5 = t4 * d
t6 = t3 + t5
```



Local Optimization

- **Finding Local Common Subexpressions**
  - Common subexpressions can be detected by noticing, as a new node ***M*** is about to be added, whether there is an existing node ***N*** with the same children, in the same order, and with the same operator
  - **Example:** A DAG for the following block

```
a = b + c
b = a - d
c = b + c
d = a - d
```



Local Optimization

- **Finding Local Common Subexpressions**
  - Since there are only three non-leaf nodes in the DAG of the previous example, the basic block can be replaced by a block **with only three statements**
  - In fact, if ***b*** is not **live on exit** from the block, then we do not need to compute that variable, and can use ***d***
  - The block then becomes

```
a = b + c
d = a - d
c = d + c
```

  - **However, if both *b* and *d* are live on exit, then a fourth statement must be used to copy the value from one to the other**

Local Optimization

- **Finding Local Common Subexpressions**
  - **Example**
    - **The following DAG does not exhibit any common subexpressions**

```
a = b + c
b = b - d
c = c + d
e = b + c
```