컴파일러의 최적화 전략 이해(1)

김성익(noerror@hitel.net)
2005.04.01

개요

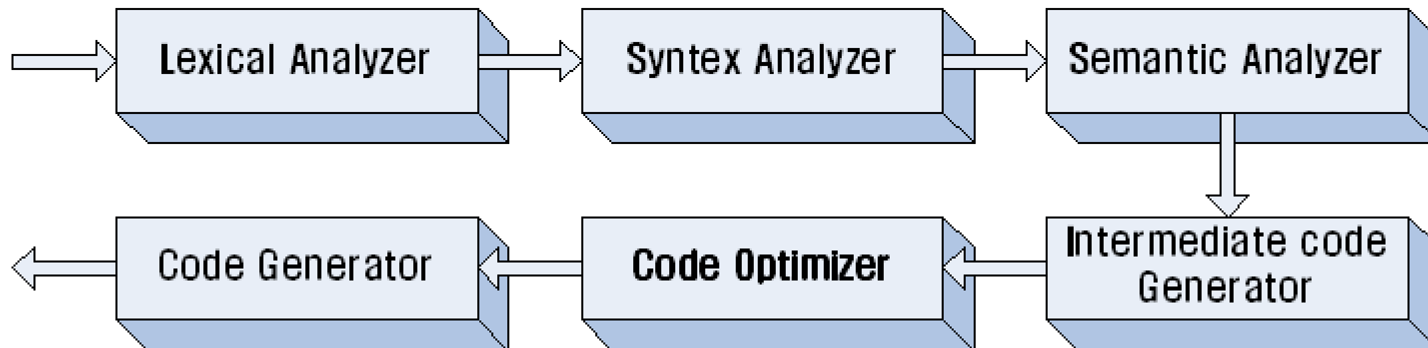
- 빠른 코드, 가독성이 좋은 소스
- 빠른 코드, 컴파일러가 좋아하는 소스

- 컴파일러 최적화에 따른 프로그래밍 전략

컴파일러

- 컴파일러의 일반적인 기능

- 언어 컴파일링의 단계



최적화

- 코드 최적화의 구성

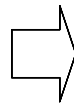
Common Subexpression Elimination (1)

- 항상 같은 결과는 내는 수식을 제거한다.
  수식이 중복되더라도 가독성 좋은 코드를 작성한다

```
void common_subexpressions(int i, int k)
{
    int a, b;

    a = i * k;
    b = i * k + 4;

    printf("%d %d", a, b);
}
```



```
8B 44 24 04        mov     eax,dword ptr [esp+4]
0F AF 44 24 08     imul    eax,dword ptr [esp+8]
00 00              add     byte ptr [eax],al
24 08              and     al,8
8D 48 04           lea     ecx,[eax+4]
51                 push    ecx
50                 push    eax
68 30 70 40 00     push    407030h
E8 18 00 00 00     call    00401030
83 C4 0C           add     esp,0Ch
C3                 ret
```

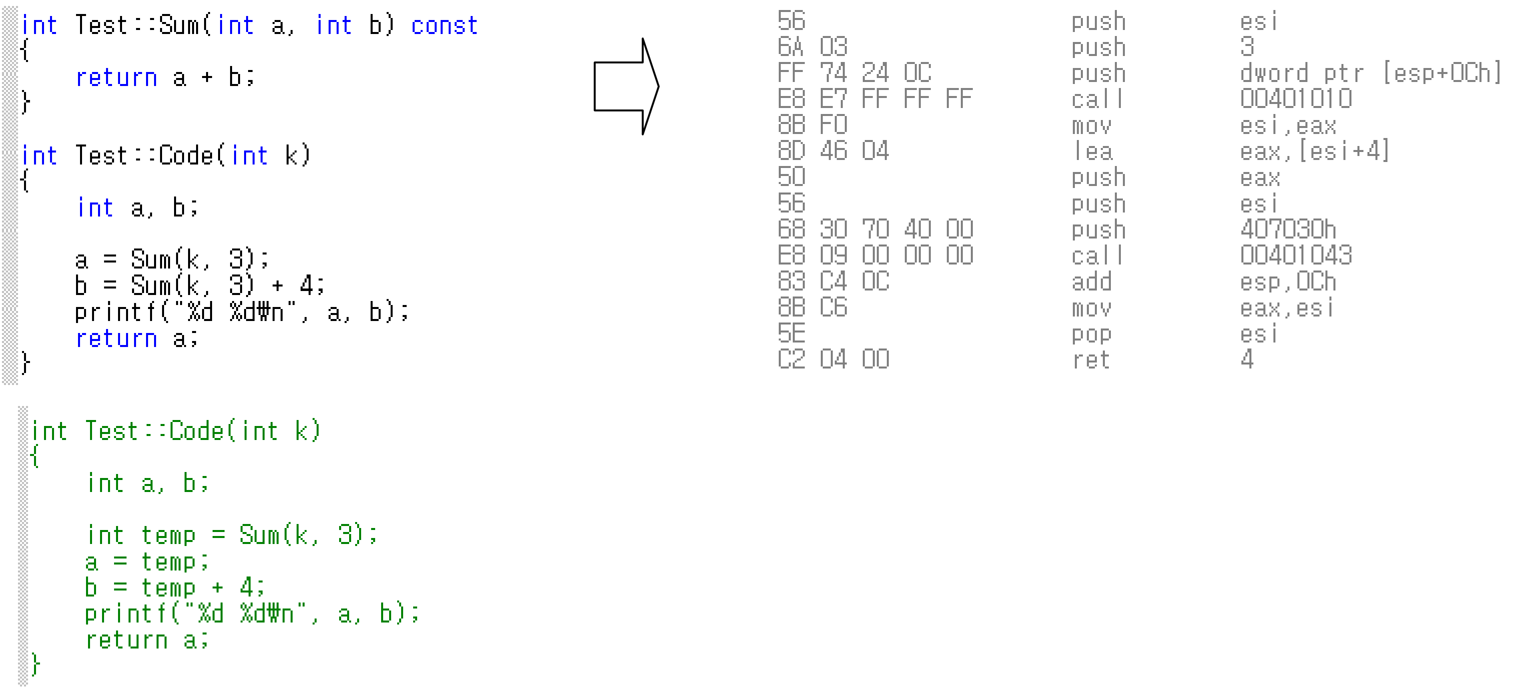
```
void common_subexpressions(int i, int k)
{
    int a, b;

    int temp = i * k;
    a = temp;
    b = temp + 4;

    printf("%d %d", a, b);
}
```

Common Subexpression Elimination (2)

- 왜 상수함수를 사용하면 좋나요 ?
  결과가 동일하기 때문에 함수 필요한 경우 호출 횟수를 줄인다



Copy Propagation

- 결과의 차이가 없고, 연산을 줄이는 경우 변수를 치환한다
  최적화를 위해서 일부러 변수를 줄일 필요는 없다.

```
int _copy_propagation(int x)
{
    int a, b;

    a = x;
    b = a + 8;
    return b;
}
```

```
int _copy_propagation(int x)
{
    int b;

    b = x + 8;
    return b;
}
```

Constant Propagation

- 변수가 상수를 담고 있는 경우 상수로 치환한다

```
int _constant_propagation(int x)
{
    int a, b;

    a = 2321;
    b = a + x;
    return b;
}
```

```
int _constant_propagation(int x)
{
    int b;

    b = 2321 + x;
    return b;
}
```

Algebraic Identities

- 결과가 명확한 수식은 가벼운 연산으로 치환한다.
  의미상 필요한 공식이라면 일부러 제거하거나 변형하지 않아도 된다.

```
void _use_of_algebraic_identities(int x)
{
    int a = x + 0;
    int b = x * 2;
    int c = 1 * x;
    int d = x / 1;
}
```

```
void _use_of_algebraic_identities(int x)
{
    int a = x;
    int b = x + x;
    int c = x;
    int d = x;
}
```

DeadCode Elimination

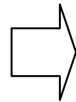
- 기능을 하지 않는 코드는 제거한다.
  참조되지 않는 함수, 코드 모두

```
int _deadcode(int d)
{
	if (d == 0 && d == 1)
	{
		return 2;
	}

	printf("%d\n", d);

	return d + 2;

	return 8;
}
```



```
68 30 70 40 00     push    407030h
00 E8              add     al,ch
E8 15 01 00 00     call    00401125
00 00              add     byte ptr [eax],al
8D 46 02           lea     eax,[esi+2]
59                 pop     ecx
5E                 pop     esi
C3                 ret
```

```
int _deadcode(int d)
{
	printf("%d\n", d);
	return d + 2;
}
```

CodeMotion

- 루프 안의 코드 중 변화가 없다고 판단되는 코드는 루프 밖으로 이동
  가독성을 헤치면서 복잡한 연산을 루프 밖으로 이동시킬 필요가 없다

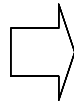
```
void _codemotion(int dx)
{
    int i, offset;

    for(i=0; i<10; i++)
    {
        offset = dx * 10 + 3;
        printf("%d\n", offset + i);
    }
}
```

```
void _codemotion(int dx)
{
    int i, offset;

    offset = dx * 10 + 3;

    for(i=0; i<10; i++)
    {
        printf("%d\n", offset + i);
    }
}
```



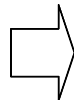
```
00401005 57             push    edi
00401006 33 F6          xor     esi,esi
00401008 8D 04 80       lea     eax,[eax+eax*4]
0040100B 8D 7C 00 03    lea     edi,[eax+eax+3]
0040100F 8D 04 37       lea     eax,[edi+esi]
00401012 50             push    eax
00401013 68 30 70 40 00 push    407030h
00401018 E8 EA 00 00 00 call    00401107
0040101D 46             inc     esi
0040101E 59             pop     ecx
0040101F 83 FE 0A       cmp     esi,0Ah
00401022 59             pop     ecx
00401023 7C EA          jl      0040100F
00401025 5F             pop     edi
00401026 5E             pop     esi
00401027 C3             ret
```

Induction Variable

- 루프안에서 일정하게 증가하는 값은 상수 덧셈으로 치환한다.
  치환이 명백한 경우 가독성을 해치면서 값싼 연산자로 치환할 필요가 없다

```c
void _induction_variables(int dx)
{
    int i, value;
    for(i=0; i<10; i++)
    {
        value = i * dx / 10;
        printf("%d₩n", value);
    }
}
```

```c
void _induction_variables(int dx)
{
    int i, value;
    int t = 0;
    for(i=0; i<10; i++, t+=dx)
    {
        value = t / 10;
        printf("%d₩n", value);
    }
}
```



```
00401002 6A 0A              push    0Ah
00401004 33 F6              xor     esi,esi
00401006 5F                 pop     edi
00401007 8B C6              mov     eax,esi
00401009 6A 0A              push    0Ah
0040100B 99                 cdq
0040100C 59                 pop     ecx
0040100D F7 F9              idiv    eax,ecx
0040100F 50                 push    eax
00401010 68 30 70 40 00     push    407030h
00401015 E8 BE 00 00 00     call    004010D8
0040101A 03 74 24 14        add     esi,dword ptr [esp+14h]
0040101E 59                 pop     ecx
0040101F 4F                 dec     edi
00401020 59                 pop     ecx
00401021 75 E4              jne     00401007
00401023 5F                 pop     edi
00401024 5E                 pop     esi
00401025 C3                 ret
```

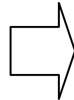
Loop Unrolling

- 짧은 루프에서 비교문의 비중이 너무 큰 경우 루프를 Unroll한다.
  똑똑한 컴파일러는 해주려나 ?

```
int loop_unrolling(int * x, int n)
{
    int i = 0, a = 0;

    do
    {
        a += x[i];
        i += 4;
    }   while(i < n);

    return a;
}
```



```
int loop_unrolling(int * x, int n)
{
    int i = 0, a = 0;

    while(i < n - 8)
    {
        a += x[i];
        a += x[i+4];
        i += 8;
    }

    do
    {
        a += x[i];
        i += 4;
    }   while(i < n);

    return a;
}
```

중간 결과

- 최적화를 해야 하는 것, 하지 않아도 되는 것
- VS .net 전역 최적화