

Code Optimization

Code Optimization

■ Code Optimization

- ✓ 실행 파일이 수행 될 때 속도와 메모리 사용 측면에서 효율적으로 동작하는 코드를 생성하는 소프트웨어 기법
- ✓ 코드 최적화는 3가지 측면을 고려함
 - 속도
 - 메모리 요구량
 - 레지스터 사용량
- ✓ 중요성
 - 컴파일 시간은 중요하지 않으나 컴파일 된 프로그램의 수행시간은 중요
 - 컴파일 된 프로그램은 수행 횟수가 정해져 있지 않음(무한대)
 - 실제로 GCC의 80~90%가 최적화를 위한 코드임.

Code Optimization

- Code Optimization

예제코드 1

```
/* original source code*/
void code_optimize1(volatile int *v1, volatile int *v2){
    *v1 += *v2;
    *v1 += *v2;
}

/* optimize version */
void code_optimize2(volatile int *v1, volatile int *v2){
    *v1 += 2* *v2;
}
```

Code Optimization

■ 예제 코드 비교

- ✓ V1의 값에 v2의 값을 2회 add연산하는 함수
- ✓ 메모리 참조 횟수 비교(최소 참조 횟수)
 - code_optimize1()
 - *ip 읽기 횟수 : 2회
 - *jp 읽기 횟수 : 2회
 - *ip 쓰기 횟수 : 2회
 - code_optimize2()
 - *ip 읽기 횟수 : 1회
 - *jp 읽기 횟수 : 1회
 - *ip 쓰기 횟수 : 1회

Code Optimization

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o code_optimize code_optimize.c

```
root@parkhk-redhat9:~/embedded_board/optimize/code_optimize_dir/dir_code_optimize
0000844c <code_optimize1>:
844c: e1a0c00d      mov     ip, sp
8450: e92dd800      stmdb  sp!, {fp, ip, lr, pc}
8454: e24cb004      sub    fp, ip, #4      ; 0x4
8458: e24dd008      sub    sp, sp, #8      ; 0x8
845c: e50b0010      str    r0, [fp, #-16]
8460: e50b1014      str    r1, [fp, #-20]
8464: e51b0010      ldr    r0, [fp, #-16]
8468: e51b3010      ldr    r3, [fp, #-16]
846c: e51b2014      ldr    r2, [fp, #-20]
8470: e5931000      ldr    r1, [r3]
8474: e5923000      ldr    r3, [r2]
8478: e0813003      add    r3, r1, r3
847c: e5803000      str    r3, [r0]
8480: e51b0010      ldr    r0, [fp, #-16]
8484: e51b3010      ldr    r3, [fp, #-16]
8488: e51b2014      ldr    r2, [fp, #-20]
848c: e5931000      ldr    r1, [r3]
8490: e5923000      ldr    r3, [r2]
8494: e0813003      add    r3, r1, r3
8498: e5803000      str    r3, [r0]
849c: e91ba800      ldmdb  fp, {fp, sp, pc}
112,0-1 398
```

함수명

r0와 r1으로 넘어온 argument를 stack에 저장

Argument가 저장되어 있는 Stack의 주소를 r0, r3, r2로 load

Register 들을 이용하여 2회 add 연산을 수행. 수행 시 각각의 결과 값을 메모리에 저장(같은 영역에 2회 저장)

Code Optimization

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o code_optimize code_optimize.c

```
000084a0 <code_optimize2>:
84a0: e1a0c00d mov ip, sp
84a4: e92dd800 stmdb sp!, {fp, ip, lr, pc}
84a8: e24cb004 sub fp, ip, #4 ; 0x4
84ac: e24dd008 sub sp, sp, #8 ; 0x8
84b0: e50b0010 str r0, [fp, -#16]
84b4: e50b1014 str r1, [fp, -#20]
84b8: e51b0010 ldr r0, [fp, -#16]
84bc: e51b1010 ldr r1, [fp, -#16]
84c0: e51b3014 ldr r3, [fp, -#20]
84c4: e5933000 ldr r3, [r3]
84c8: e1a02083 mov r2, r3, lsl #1
84cc: e5913000 ldr r3, [r1]
84d0: e0833002 add r3, r3, r2
84d4: e5803000 str r3, [r0]
84d8: e91ba800 ldmdb fp, {fp, sp, pc}
135,0-1 468
```

함수명

r0와 r1으로 넘어온 argument를 stack에 저장

Argument가 저장되어 있는 Stack의 주소를 r0, r3, r2로 load

r3의 값을 1회 left shift를 통해 code_optimize1과 같은 결과 값 산출

Code Optimization

- O0 옵션을 이용한 컴파일 결과
 - ✓ objdump 결과 ■ 부분이 최적화 전후 코드의 변화를 나타내고 있음
 - ✓ code_optimize1()
 - 명령어 수행횟수
 - ldr : 10회
 - add : 2회
 - str : 2회
 - ✓ code_optimize2()
 - 명령어 수행횟수
 - ldr : 2회
 - add : 1회
 - mov : 1회
 - str : 1회

Code Optimization

■ O0 옵션을 이용한 컴파일 결과

✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 23000nsec, 16000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_code_optimization]$ ./code_optimize  
  
code_optimize1()  
Before v1=1, v2=2  
Iteration : 101  
After v1=405, v2=2  
0sec 23000nsec  
  
code_optimize1()  
Before v1=1, v2=2  
Iteration : 101  
After v1=405, v2=2  
0sec 16000nsec  
  
[root@ez-x5 dir_code_optimization]$ █
```


Code Optimization

- O1 옵션을 이용한 컴파일 결과

- ✓ armv5l-linux-gcc -O1 -lrt -o code_optimize code_optimize.c


```
root@parkhk-redhat9:~/embedded_board/optimize/code_optimize_dir/dir_code_optimize
0000844c <code_optimize1>:
844c: e5903000 ldr r3, [r0]
8450: e5912000 ldr r2, [r1]
8454: e0833002 add r3, r3, r2
8458: e5803000 str r3, [r0]
845c: e5903000 ldr r3, [r0]
8460: e5912000 ldr r2, [r1]
8464: e0833002 add r3, r3, r2
8468: e5803000 str r3, [r0]
846c: e12fff1e bx lr

00008470 <code_optimize2>:
8470: e5912000 ldr r2, [r1]
8474: e5903000 ldr r3, [r0]
8478: e0833082 add r3, r3, r2, lsl #1
847c: e5803000 str r3, [r0]
8480: e12fff1e bx lr

112,0-1 518
```

Code Optimization

■ O1 옵션을 이용한 컴파일 결과

- ✓ objdump 결과  부분이 최적화 전후 코드의 변화를 나타내고 있음
- ✓ code_optimize1()
 - 결과값 산출 명령어 수행횟수
 - ldr : 4회
 - add : 2회
 - str : 2회
- ✓ code_optimize2()
 - 결과값 산출 명령어 수행횟수
 - ldr : 2회
 - add : 1회
 - str : 1회

Code Optimization

■ 이 옵션을 이용한 컴파일 결과

✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 15000nsec, 7000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_code_optimization]$ ./code_optimize  
  
code_optimize1()  
Before v1=1, v2=2  
Iteration : 101  
After v1=405, v2=2  
0sec 15000nsec  
  
code_optimize1()  
Before v1=1, v2=2  
Iteration : 101  
After v1=405, v2=2  
0sec 7000nsec  
  
[root@ez-x5 dir_code_optimization]$ █
```

Code Optimization

■ O2 옵션을 이용한 컴파일 결과

✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 15000nsec, 7000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_code_optimization]$ ./code_optimize  
  
code_optimize1()  
Before v1=1, v2=2  
Iteration : 101  
After v1=405, v2=2  
0sec 14000nsec  
  
code_optimize1()  
Before v1=1, v2=2  
Iteration : 101  
After v1=405, v2=2  
0sec 7000nsec  
  
[root@ez-x5 dir_code_optimization]$ █
```

O2 옵션을 이용하여 컴파일 한 결과 O1 옵션을 이용한 컴파일 결과와 같았음(code_optimize1과 2함수).

Code Optimization

■ O3옵션을 이용한 컴파일 결과

✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 15000nsec, 7000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_code_optimization]$ ./code_optimize  
  
code_optimize1()  
Before v1=1, v2=2  
Iteration : 101  
After v1=405, v2=2  
0sec 12000nsec  
  
code_optimize1()  
Before v1=1, v2=2  
Iteration : 101  
After v1=405, v2=2  
0sec 4000nsec  
  
[root@ez-x5 dir_code_optimization]$ █
```

O3옵션을 이용하여 컴파일 한 결과 O1옵션을 이용한 컴파일 결과와 같았음(code_optimize1과 2함수).

Code Optimization

■ Code Optimization

예제코드 2

```
void code_opt_swap1(int *v1, int *v2){  
  
    int tmp;  
  
    tmp = *v1;  
    *v1 = *v2;  
    *v2 = tmp;  
}
```

```
void code_opt_swap2(int *v1, int *v2){  
  
    (*v1) ^ = (*v2) ^ = (*v1) ^ = (*v2);  
}
```

Code Optimization

■ 예제 코드 비교

✓ V1의 값과 v2의 값을 swap하는 함수

✓ File size

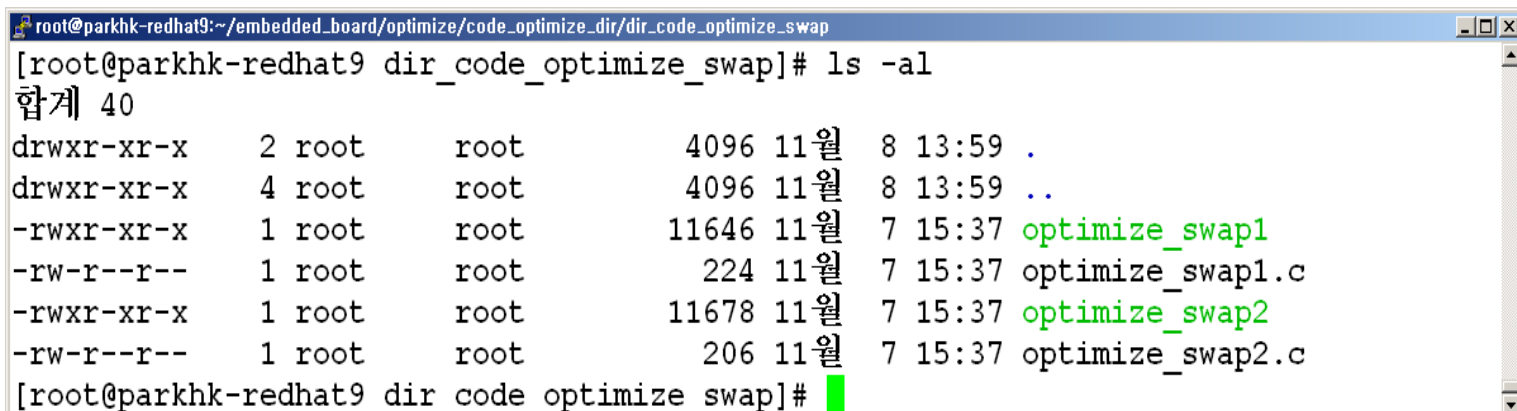
■ 소스 파일

- code_opt_swap1() : 224byte
- code_opt_swap2() : 206byte

■ 실행 파일

- code_opt_swap1() : 11646byte
- code_opt_swap2() : 11678byte

■ 컴파일 후 파일의 사이즈는 소스 파일의 라인수에 무관할 수 있음



```
root@parkhk-redhat9:~/embedded_board/optimize/code_optimize_dir/dir_code_optimize_swap
[root@parkhk-redhat9 dir_code_optimize_swap]# ls -al
합계 40
drwxr-xr-x  2 root    root      4096 11월  8 13:59 .
drwxr-xr-x  4 root    root      4096 11월  8 13:59 ..
-rwxr-xr-x  1 root    root     11646 11월  7 15:37 optimize_swap1
-rw-r--r--  1 root    root      224 11월  7 15:37 optimize_swap1.c
-rwxr-xr-x  1 root    root     11678 11월  7 15:37 optimize_swap2
-rw-r--r--  1 root    root      206 11월  7 15:37 optimize_swap2.c
[root@parkhk-redhat9 dir_code_optimize_swap]#
```

Code Optimization

■ O0 옵션을 이용한 컴파일 결과

- ✓ `armv5l-linux-gcc -O0 -lrt -o optimize_swap optimize_swap.c`

```
root@parkhk-redhat9:~/embedded_board/optimize/code_optimize_dir/dir_code_optimize_swap
0000844c <code_opt_swap1>:
844c: e1a0c00d    mov     ip, sp
8450: e92dd800    stmdb  sp!, {fp, ip, lr, pc}
8454: e24cb004    sub    fp, ip, #4      ; 0x4
8458: e24dd00c    sub    sp, sp, #12    ; 0xc
845c: e50b0010    str    r0, [fp, #-16]
8460: e50b1014    str    r1, [fp, #-20]
8464: e51b3010    ldr    r3, [fp, #-16]
8468: e5933000    ldr    r3, [r3]
846c: e50b3018    str    r3, [fp, #-24]
8470: e51b2010    ldr    r2, [fp, #-16]
8474: e51b3014    ldr    r3, [fp, #-20]
8478: e5933000    ldr    r3, [r3]
847c: e5823000    str    r3, [r2]
8480: e51b2014    ldr    r2, [fp, #-20]
8484: e51b3018    ldr    r3, [fp, #-24]
8488: e5823000    str    r3, [r2]
848c: e91ba800    ldmdb fp, {fp, sp, pc}

112,0-1 378
```


Code Optimization

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o optimize_swap optimize_swap.c

```
root@parkhk-redhat9:~/embedded_board/optimize/code_optimize_dir/dir_code_optimize_swap
00008490 <code_opt_swap2>:
8490: e1a0c00d mov ip, sp
8494: e92dd830 stmdb sp!, {r4, r5, fp, ip, lr, pc}
8498: e24cb004 sub fp, ip, #4 ; 0x4
849c: e24dd008 sub sp, sp, #8 ; 0x8
84a0: e50b0018 str r0, [fp, -#24]
84a4: e50b101c str r1, [fp, -#28]
84a8: e51be018 ldr lr, [fp, -#24]
84ac: e51b4018 ldr r4, [fp, -#24]
84b0: e51b501c ldr r5, [fp, -#28]
84b4: e51b001c ldr r0, [fp, -#28]
84b8: e51bc018 ldr ip, [fp, -#24]
84bc: e51b3018 ldr r3, [fp, -#24]
84c0: e51b201c ldr r2, [fp, -#28]
84c4: e5931000 ldr r1, [r3]
84c8: e5923000 ldr r3, [r2]
84cc: e0213003 eor r3, r1, r3
84d0: e1a02003 mov r2, r3
84d4: e58c2000 str r2, [ip]
84d8: e5903000 ldr r3, [r0]
84dc: e0233002 eor r3, r3, r2
84e0: e1a02003 mov r2, r3
84e4: e5852000 str r2, [r5]
84e8: e5943000 ldr r3, [r4]
84ec: e0233002 eor r3, r3, r2
84f0: e58e3000 str r3, [lr]
84f4: e91ba830 ldmdb fp, {r4, r5, fp, sp, pc}
```

함수명

3회 XOR을 수행하기 위해 여러 번의 ldr을 수행. 여러 개의 레지스터들을 사용. 결과적으로 code_opt_swap1 함수보다 비효율적인 코드가 생성되었다.

132,5 458

Code Optimization

- O0 옵션을 이용한 컴파일 결과
 - ✓ objdump 결과 ■ 부분이 최적화 전후 코드의 변화를 나타내고 있음
 - ✓ code_opt_swap1()
 - 명령어 수행횟수
 - ldr : 7회
 - str : 5회
 - ✓ code_opt_swap2()
 - 명령어 수행횟수
 - ldr : 11회
 - eor : 3회
 - mov : 2회
 - str : 5회

Code Optimization

- O0 옵션을 이용한 컴파일 결과

- ✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 14616000nsec, 17119000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_code_optimization]$ ./optimize_swap  
code_opt_swap1()  
Before v1=1, v2=2  
Iteration : 10001  
After v1=2, v2=1  
0sec 14616000nsec  
  
code_opt_swap2()  
Before v1=1, v2=2  
Iteration : 10001  
After v1=2, v2=1  
0sec 17119000nsec  
  
[root@ez-x5 dir_code_optimization]$ █
```

Code Optimization

■ O3 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O3 -lrt -o optimize_swap optimize_swap.c

```
root@parkhk-redhat9:~/embedded_board/optimize/code_optimize_dir/dir_code_optimize_swap
0000859c <code_opt_swap1>:
859c: e590c000 ldr ip, [r0]
85a0: e5912000 ldr r2, [r1]
85a4: e5802000 str r2, [r0]
85a8: e581c000 str ip, [r1]
85ac: e12fff1e bx lr

000085b0 <code_opt_swap2>:
85b0: e5912000 ldr r2, [r1]
85b4: e590c000 ldr ip, [r0]
85b8: e02c3002 eor r3, ip, r2
85bc: e5803000 str r3, [r0]
85c0: e591c000 ldr ip, [r1]
85c4: e02c2003 eor r2, ip, r3
85c8: e5812000 str r2, [r1]
85cc: e590c000 ldr ip, [r0]
85d0: e02c1002 eor r1, ip, r2
85d4: e5801000 str r1, [r0]
85d8: e12fff1e bx lr
```


함수명

함수명

218,0-1 88%

Code Optimization

■ O3 옵션을 이용한 컴파일 결과

- ✓ objdump 결과  부분이 최적화 전후 코드의 변화를 나타내고 있음
- ✓ code_opt_swap1()
 - 명령어 수행횟수
 - ldr : 2회
 - str : 2회
- ✓ code_opt_swap2()
 - 명령어 수행횟수
 - ldr : 4회
 - eor : 3회
 - str : 4회

Code Optimization

■ O3옵션을 이용한 컴파일 결과

✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 1514000nsec, 2024000nsec의 시간이 걸렸음
- O0옵션으로 컴파일한 경우에 비하여 약 10정도 성능 향상을 보이고 있음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_code_optimization]$ ./optimize_swap  
code_opt_swap1()  
Before v1=1, v2=2  
Iteration : 10001  
After v1=2, v2=1  
0sec 1514000nsec  
  
code_opt_swap2()  
Before v1=1, v2=2  
Iteration : 10001  
After v1=2, v2=1  
0sec 2024000nsec  
  
[root@ez-x5 dir_code_optimization]$ █
```

Code Optimization

■ Code Optimization

- ✓ 컴파일 후 명령어 수행 횟수를 최소화 함으로써 성능을 높일 수 있음.
- ✓ 일반적으로는 길이가 짧은 코드가 적은 명령어를 수행하지만 항상 그러하지는 않음.
- ✓ 의도적으로 delay를 주는 경우가 아니라면 -O3 옵션이 최적화에 큰 역할을 할 수 있음

코드 최적화 기법

- Loop Fusion
- Loop Unrolling
- Loop Invariant Code Motion
- Strength Reduction
- Count up to Zero

Loop Fusion

■ Loop Fusion

- ✓ Loop 최적화 기법
- ✓ Loop Jamming 이라고도 함
- ✓ 반복문은 다른 연산에 비하여 상대적으로 많은 cycle을 요구
- ✓ 반복문의 개수를 최소화 하는 것이 성능향상에 도움이 됨
- ✓ Fusion이 가능한 반복문에 대하여 루프를 하나로 결합

Loop Fusion

- Loop Fusion

예제코드

```
/* original source code*/

int array_v1[ARRAY_SIZE];
int array_v2[ARRAY_SIZE];

int loop_fusion1(void){

    int i;
    for(i=0;i<ARRAY_SIZE;i+2){
        array_v1[i] = i;
    }

    for(i=0;i<ARRAY_SIZE;i+2){
        array_v2[i] = i;
    }
}
```

Loop Fusion

- Loop Fusion

예제코드

```
/* Loop Fusion technique */  
  
int array_v1[ARRAY_SIZE];  
int array_v2[ARRAY_SIZE];  
  
int loop_fusion2(void){  
  
    int i;  
    for(i=0;i<ARRAY_SIZE;i+2){  
  
        array_v1[i] = i;  
        array_v2[i] = i;  
    }  
}
```

Loop Fusion

■ 예제 코드 비교

- ✓ Loop 반복 횟수
 - loop_fusion1
 - ARRAY_SIZE * 2
 - loop_fusion2
 - ARRAY_SIZE

- ✓ 불필요한 루프를 줄임으로써 상대적으로 적은 cycle안에 코드를 수행

- ✓ 특이 사항
 - 전역 변수 사용시 명시적으로 초기화를 해주지 않으면 zeroed page 에 mapping된다. 그래서 전역변수를 처음 접근하게 되면 page fault를 유발하게 되며 그때서야 메모리를 할당 받게 된다. 최적화 코드에 공정성을 기하기 위해 전역 변수들은 미리 한번 초기화를 수행 하였다. 본 모듈에서 사용하는 모든 소스는 배열 초기화를 미리 한 후 테스트 하였다.

Loop Fusion

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o loop_fusion loop_fusion.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_fusion_dir
113 0000844c <loop_fusion1>:
114 844c: e1a0c00d mov ip, sp
115 8450: e92dd800 stmdb sp!, {fp, ip, lr, pc}
116 8454: e24cb004 sub fp, ip, #4 ; 0x4
117 8458: e24dd004 sub sp, sp, #4 ; 0x4
118 845c: e3a03000 mov r3, #0 ; 0x0
119 8460: e50b3010 str r3, [fp, #-16]
120 8464: e51b2010 ldr r2, [fp, #-16]
121 8468: e3a03a02 mov r3, #8192 ; 0x2000
122 846c: e2833003 add r3, r3, #3 ; 0x3
123 8470: e1520003 cmp r2, r3
124 8474: da000000 ble 847c <loop_fusion1+0x30>
125 8478: ea000007 b 849c <loop_fusion1+0x50>
126 847c: e59f1060 ldr r1, [pc, #96] ; 84e4 <loop_fusion1+0x98>
127 8480: e51b2010 ldr r2, [fp, #-16]
128 8484: e51b3010 ldr r3, [fp, #-16]
129 8488: e7813102 str r3, [r1, r2, lsl #2]
130 848c: e51b3010 ldr r3, [fp, #-16]
131 8490: e2833001 add r3, r3, #1 ; 0x1
132 8494: e50b3010 str r3, [fp, #-16]
133 8498: eafffff1 b 8464 <loop_fusion1+0x18>
```

함수명

첫 번째 for 문

133,1 418

Loop Fusion

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o loop_fusion loop_fusion.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_fusion_dir
134 849c: e3a03000
135 84a0: e50b3010
136 84a4: e51b2010
137 84a8: e3a03a02
138 84ac: e2833003
139 84b0: e1520003
140 84b4: da000000
141 84b8: ea000007
142 84bc: e59f1024
143 84c0: e51b2010
144 84c4: e51b3010
145 84c8: e7813102
146 84cc: e51b3010
147 84d0: e2833001
148 84d4: e50b3010
149 84d8: eafffff1
150 84dc: e1a00003
151 84e0: e91ba800
152 84e4: 000188a0
153 84e8: 00010880
154
```

```
mov r3, #0 ; 0x0
str r3, [fp, #-16]
ldr r2, [fp, #-16]
mov r3, #8192 ; 0x2000
add r3, r3, #3 ; 0x3
cmp r2, r3
ble 84bc <loop_fusion1+0x70>
b 84dc <loop_fusion1+0x90>
ldr r1, [pc, #36] ; 84e8 <loop_fusion1+0x9c>
ldr r2, [fp, #-16]
ldr r3, [fp, #-16]
str r3, [r1, r2, lsl #2]
ldr r3, [fp, #-16]
add r3, r3, #1 ; 0x1
str r3, [fp, #-16]
b 84a4 <loop_fusion1+0x58>
mov r0, r3
ldmdb fp, {fp, sp, pc}
andeq r8, r1, r0, lsr #17
andeq r0, r1, r0, lsl #17
```

함수명:
loop_fusion1()

두 번째
for 문

첫 번째 for문에
서 수행하던 내용
과 같음.

154,0-1 498

Loop Fusion

- O0 옵션을 이용한 컴파일 결과

- ✓ armv5l-linux-gcc -O0 -lrt -o loop_fusion loop_fusion.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_fusion_dir
155 000084ec <loop_fusion2>:
156 84ec: e1a0c00d mov ip, sp
157 84f0: e92dd800 stmdb sp!, {fp, ip, lr, pc}
158 84f4: e24cb004 sub fp, ip, #4 ; 0x4
159 84f8: e24dd004 sub sp, sp, #4 ; 0x4
160 84fc: e3a03000 mov r3, #0 ; 0x0
161 8500: e50b3010 str r3, [fp, #-16]
162 8504: e51b2010 ldr r2, [fp, #-16]
163 8508: e3a03a02 mov r3, #8192 ; 0x2000
164 850c: e2833003 add r3, r3, #3 ; 0x3
165 8510: e1520003 cmp r2, r3
166 8514: da000000 ble 851c <loop_fusion2+0x30>
167 8518: ea00000b b 854c <loop_fusion2+0x60>
168 851c: e59f1030 ldr r1, [pc, #48] ; 8554 <loop_fusion2+0x68>
169 8520: e51b2010 ldr r2, [fp, #-16]
```

Loop Fusion

- O0 옵션을 이용한 컴파일 결과

- ✓ `armv5l-linux-gcc -O0 -lrt -o loop_fusion loop_fusion.c`

```
root@parkh-redhat9:~/embedded_board/optimize/loop_fusion_dir
170 8524: e51b3010 ldr r3, [fp, -#16]
171 8528: e7813102 str r3, [r1, r2, lsl #2]
172 852c: e59f1024 ldr r1, [pc, #36] ; 8558 <loop_fusion2+0x6c>
173 8530: e51b2010 ldr r2, [fp, -#16]
174 8534: e51b3010 ldr r3, [fp, -#16]
175 8538: e7813102 str r3, [r1, r2, lsl #2]
176 853c: e51b3010 ldr r3, [fp, -#16]
177 8540: e2833001 add r3, r3, #1 ; 0x1
178 8544: e50b3010 str r3, [fp, -#16]
179 8548: eaffffed b 8504 <loop_fusion2+0x18>
180 854c: e1a00003 mov r0, r3
181 8550: e91ba800 ldmdb fp, {fp, sp, pc}
182 8554: 000188a0 andeq r8, r1, r0, lsr #17
183 8558: 00010880 andeq r0, r1, r0, lsl #17
184
```



함수명:
loop_fusion2()

중복되는 loop
수행을 줄임으
로써 코드가 간
결화 되었음

184,0-1 618

Loop Fusion

■ O0 옵션을 이용한 컴파일 결과

- ✓  부분은 for문을 수행하기 위한 arm assembly이고  부분은 배열에 값을 초기화 해주는 부분이다.

✓ loop_fusion1()

- for loop 수행 코드
 - mov, str, ldr, add, b : 4회
 - cmp, ble : 2회
- 배열 값 초기화 수행 코드
 - ldr : 6회
 - str : 2회

✓ loop_fusion2()

- for loop 수행 코드
 - mov, str, ldr, add, b : 2회
 - cmp, ble : 1회
- 배열 값 초기화 수행 코드
 - ldr : 6회
 - str : 2회

Loop Fusion

- OO옵션을 이용한 컴파일 결과

- ✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 2382000nsec, 1098000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_loop_fusion]$ ./loop_fusion  
  
loop_fusion1()  
Iteration : 8196  
0sec 2382000nsec  
  
loop_fusion2()  
Iteration : 8196  
0sec 1098000nsec  
[root@ez-x5 dir_loop_fusion]$
```

Loop Fusion

- O3 옵션을 이용한 컴파일 결과

- ✓ armv5l-linux-gcc -O3 -lrt -o loop_fusion loop_fusion.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_fusion_dir
000085ac <loop_fusion1>:
85ac: e59f1034 ldr r1, [pc, #52] ; 85e8 <loop_fusion1+0x3c>
85b0: e3a00a02 mov r0, #8192 ; 0x2000
85b4: e3a03000 mov r3, #0 ; 0x0
85b8: e2802003 add r2, r0, #3 ; 0x3
85bc: e7813103 str r3, [r1, r3, lsl #2]
85c0: e2833001 add r3, r3, #1 ; 0x1
85c4: e1530002 cmp r3, r2
85c8: daffffff ble 85bc <loop_fusion1+0x10>
85cc: e59f1018 ldr r1, [pc, #24] ; 85ec <loop_fusion1+0x40>
85d0: e3a03000 mov r3, #0 ; 0x0
85d4: e7813103 str r3, [r1, r3, lsl #2]
85d8: e2833001 add r3, r3, #1 ; 0x1
85dc: e1530002 cmp r3, r2
85e0: daffffff ble 85d4 <loop_fusion1+0x28>
85e4: e12fff1e bx lr
85e8: 00018820 andeq r8, r1, r0, lsr #16
85ec: 00010800 andeq r0, r1, r0, lsl #16
```

함수명

첫 번째 for 문

두 번째 for 문

194,0-1 828

Loop Fusion

- O3 옵션을 이용한 컴파일 결과

- ✓ armv5l-linux-gcc -O0 -lrt -o loop_fusion loop_fusion.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_fusion_dir
000085f0 <loop_fusion2>:
85f0: e59f0024 ldr r0, [pc, #36] ; 861c <loop_fusion2+0x2c>
85f4: e59f1024 ldr r1, [pc, #36] ; 8620 <loop_fusion2+0x30>
85f8: e3a0ca02 mov ip, #8192 ; 0x2000
85fc: e3a03000 mov r3, #0 ; 0x0
8600: e28c2003 add r2, ip, #3 ; 0x3
8604: e7803103 str r3, [r0, r3, lsl #2]
8608: e7813103 str r3, [r1, r3, lsl #2]
860c: e2833001 add r3, r3, #1 ; 0x1
8610: e1530002 cmp r3, r2
8614: dafffffa ble 8604 <loop_fusion2+0x14>
8618: e12fff1e bx lr
861c: 00018820 andeq r8, r1, r0, lsr #16
8620: 00010800 andeq r0, r1, r0, lsl #16
```



함수명

첫 번째 for 문

214,1 89%

Loop Fusion

■ O3옵션을 이용한 컴파일 결과

- ✓  부분은 for문을 수행하기 위한 arm assembly이고  부분은 배열에 값을 초기화 해주는 부분이다.

✓ loop_fusion1()

- for loop 수행 코드
 - mov, add: 3회
 - cmp, ble : 2회
- 배열 값 초기화 수행 코드
 - ldr : 2회
 - str : 2회

✓ loop_fusion2()

- for loop 수행 코드
 - mov, add : 2회
 - cmp, ble : 1회
- 배열 값 초기화 수행 코드
 - ldr : 2회
 - str : 2회

- ✓ 위의 명령어들이 반복횟수 만큼 수행됨

Loop Fusion

■ O3옵션을 이용한 컴파일 결과

✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 2068000nsec, 792000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_loop_fusion]$ ./loop_fusion  
  
loop_fusion1()  
Iteration : 8196  
0sec 2068000nsec  
  
loop_fusion2()  
Iteration : 8196  
0sec 792000nsec  
[root@ez-x5 dir_loop_fusion]$
```

Loop Fusion

■ Loop Fusion

- ✓ 수행 시간
 - 여러 개의 반복문을 하나로 묶음으로써 Loop수행에 따른 cycle을 절약할 수 있다
- ✓ 효과적인 cache사용으로 인한 성능상 이득을 얻을 수 있음

Loop Unrolling

■ Loop unrolling

- ✓ Loop 최적화 기법
- ✓ 반복문이 포함하고 있는 원래의 코드를 여러 번 반복하여 작성
- ✓ 반복문에서 루프 수행 횟수를 감소 시키는 기법
- ✓ 배열의 크기가 unrolling의 양의 배수가 되도록 작성
 - 배열의 크기가 unrolling양에 비례할 경우 가장 좋은 성능을 보일 수 있음
 - 양의 배수가 안될 경우 루프 바깥쪽에서 unrolling하는 것을 고려

Loop Unrolling

- Loop unrolling

예제코드

```
/* original source code */  
int array_v[ARRAY_SIZE];  
int loop_unrolling1(void){  
    int i;  
    for(i=0; i<ARRAY_SIZE ; i++){  
        array_v[i] = i;  
    }  
}
```

Loop Unrolling

- Loop unrolling

예제코드

```
/*Loop unrolling technique*/  
  
int array_v[ARRAY_SIZE];  
  
int loop_unrolling2(void){  
  
    int i;  
  
    for(i=0; i<ARRAY_SIZE ; i+=2){  
        array_v[i] = i;  
        array_v[i+1] = i+1;  
    }  
}
```

Loop Unrolling

■ 예제 코드 비교

- ✓ Loop 반복 횟수
 - loop_unrolling1
 - ARRAY_SIZE
 - loop_unrolling2
 - ARRAY_SIZE/2

- ✓ 등차수열의 형태를 갖는 반복문에서 배열의 초기화 수행 시 배열을 unrolling을 함으로써 반복 횟수를 줄임

Loop Unrolling

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o loop_unrolling loop_unrolling.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_unrolling_dir
0000844c <loop_unrolling1>:
844c:    e1a0c00d    mov     ip, sp
8450:    e92dd800    stmdb  sp!, {fp, ip, lr, pc}
8454:    e24cb004    sub    fp, ip, #4      ; 0x4
8458:    e24dd004    sub    sp, sp, #4     ; 0x4
845c:    e3a03000    mov    r3, #0        ; 0x0
8460:    e50b3010    str    r3, [fp, -#16]
8464:    e51b2010    ldr    r2, [fp, -#16]
8468:    e3a03a02    mov    r3, #8192     ; 0x2000
846c:    e2833003    add    r3, r3, #3     ; 0x3
8470:    e1520003    cmp    r2, r3
8474:    da000000    ble   847c <loop_unrolling1+0x30>
8478:    ea000007    b     849c <loop_unrolling1+0x50>
847c:    e59f1020    ldr    r1, [pc, #32]  ; 84a4 <loop_unrolling1+0x58>
8480:    e51b2010    ldr    r2, [fp, -#16]
8484:    e51b3010    ldr    r3, [fp, -#16]
8488:    e7813102    str    r3, [r1, r2, lsl #2]
848c:    e51b3010    ldr    r3, [fp, -#16]
8490:    e2833001    add    r3, r3, #1     ; 0x1
8494:    e50b3010    str    r3, [fp, -#16]
8498:    eafffff1    b     8464 <loop_unrolling1+0x18>
849c:    e1a00003    mov    r0, r3
84a0:    e91ba800    ldmdb fp, {fp, sp, pc}
84a4:    00010e20    andeq r0, r1, r0, lsr #28
```

113,5

178

Loop Unrolling

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o loop_unrolling loop_unrolling.c

```
000084a8 <loop_unrolling2>:
84a8: e1a0c00d    mov     ip, sp
84ac: e92dd800    stmdb  sp!, {fp, ip, lr, pc}
84b0: e24cb004    sub    fp, ip, #4 ; 0x4
84b4: e24dd004    sub    sp, sp, #4 ; 0x4
84b8: e3a03000    mov    r3, #0 ; 0x0
84bc: e50b3010    str    r3, [fp, -#16]
84c0: e51b2010    ldr    r2, [fp, -#16]
84c4: e3a03a02    mov    r3, #8192 ; 0x2000
84c8: e2833003    add    r3, r3, #3 ; 0x3
84cc: e1520003    cmp    r2, r3
84d0: da000000    ble   84d8 <loop_unrolling2+0x30>
84d4: ea000010    b     851c <loop_unrolling2+0x74>
84d8: e59f1044    ldr    r1, [pc, #68] ; 8524 <loop_unrolling2+0x7c>
84dc: e51b2010    ldr    r2, [fp, -#16]
84e0: e51b3010    ldr    r3, [fp, -#16]
84e4: e7813102    str    r3, [r1, r2, lsl #2]
84e8: e59f1034    ldr    r1, [pc, #52] ; 8524 <loop_unrolling2+0x7c>
84ec: e51b3010    ldr    r3, [fp, -#16]
84f0: e3a02004    mov    r2, #4 ; 0x4
84f4: e1a03103    mov    r3, r3, lsl #2
84f8: e0833001    add    r3, r3, r1
84fc: e0832002    add    r2, r3, r2
8500: e51b3010    ldr    r3, [fp, -#16]
8504: e2833001    add    r3, r3, #1 ; 0x1
8508: e5823000    str    r3, [r2]
850c: e51b3010    ldr    r3, [fp, -#16]
8510: e2833002    add    r3, r3, #2 ; 0x2
8514: e50b3010    str    r3, [fp, -#16]
8518: eaffffe8    b     84c0 <loop_unrolling2+0x18>
851c: e1a00003    mov    r0, r3
8520: e91ba800    ldmdb fp, {fp, sp, pc}
8524: 00010e20    andeq r0, r1, r0, lsr #28
```


함수명

8196/2회 for문을 수행

137,0-1 228

Loop Unrolling

■ O0 옵션을 이용한 컴파일 결과

✓  부분은 for문을 수행하기 위한 arm assembly임

✓ loop_unrolling1()

- for loop 수행 코드
 - mov, str, ldr, add, b : 2회
 - cmp, ble : 1회
- 배열 값 초기화 수행 코드
 - ldr : 3회
 - str : 1회
- 반복횟수
 - 8196

✓ loop_unrolling2()

- for loop 수행 코드
 - mov, str, ldr, add, b : 2회
 - cmp, ble : 1회
- 배열 값 초기화 수행 코드
 - ldr : 6회
 - str : 2회
 - mov : 2회
 - add : 3회
- 반복횟수
 - 4098회

Loop Unrolling

- OO옵션을 이용한 컴파일 결과

- ✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 1199000nsec, 637000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_loop_unrolling]$ ./loop_unrolling  
  
loop_unrolling1()  
Iteration : 8196  
0sec 1199000nsec  
  
loop_unrolling1()  
Iteration : 8196  
0sec 637000nsec  
[root@ez-x5 dir_loop_unrolling]$
```

Loop Unrolling

■ O3 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O3 -lrt -o loop_unrolling loop_unrolling.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_unrolling_dir
0000858c <loop_unrolling1>:
858c: e59f101c ldr r1, [pc, #28] ; 85b0 <loop_unrolling1+0x24>
8590: e3a00a02 mov r0, #8192 ; 0x2000
8594: e3a03000 mov r3, #0 ; 0x0
8598: e2802003 add r2, r0, #3 ; 0x3
859c: e7813103 str r3, [r1, r3, lsl #2]
85a0: e2833001 add r3, r3, #1 ; 0x1
85a4: e1530002 cmp r3, r2
85a8: daffffff ble 859c <loop_unrolling1+0x10>
85ac: e12fff1e bx lr
85b0: 00010920 andeq r0, r1, r0, lsr #18

000085b4 <loop_unrolling2>:
85b4: e59fc02c ldr ip, [pc, #44] ; 85e8 <loop_unrolling2+0x34>
85b8: e3a01a02 mov r1, #8192 ; 0x2000
85bc: e2810003 add r0, r1, #3 ; 0x3
85c0: e3a02000 mov r2, #0 ; 0x0
85c4: e1a0100c mov r1, ip
85c8: e78c2102 str r2, [ip, r2, lsl #2]
85cc: e2823001 add r3, r2, #1 ; 0x1
85d0: e2822002 add r2, r2, #2 ; 0x2
85d4: e1520000 cmp r2, r0
85d8: e5813004 str r3, [r1, #4]
85dc: e2811008 add r1, r1, #8 ; 0x8
85e0: daffffff ble 85c8 <loop_unrolling2+0x14>
85e4: e12fff1e bx lr
85e8: 00010920 andeq r0, r1, r0, lsr #18
```

함수명

함수명


배열 초기화중
index로 사용된
값을 loop에서
cmp시 사용할
변수로 재사용함
으로써 최적화
되었음.

186,0-1

608

Loop Unrolling

■ O3옵션을 이용한 컴파일 결과

✓  부분은 for문을 수행하기 위한 arm assembly임

✓ loop_fusion1()

- for loop 수행 코드
 - mov, add : 2회
 - cmp, ble : 1회
- 배열 값 초기화 수행 코드
 - ldr : 1회
 - str : 1회
- 반복횟수
 - 8196

✓ loop_fusion2()

- for loop 수행 코드
 - mov: 2회
 - add, cmp, ble : 1회
- 배열 값 초기화 수행 코드
 - ldr : 1회
 - str : 2회
 - mov : 1회
 - add : 3회
- 반복횟수
 - 4098회

Loop Unrolling

- O3옵션을 이용한 컴파일 결과

- ✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 968000nsec, 441000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_loop_unrolling]$ ./loop_unrolling  
  
loop_unrolling1()  
Iteration : 8196  
0sec 968000nsec  
  
loop_unrolling1()  
Iteration : 8196  
0sec 441000nsec  
[root@ez-x5 dir_loop_unrolling]$
```

Loop Unrolling

- unrolling 개수와 최적화 그리고 프로그램 사이즈의 관계

✓ 시간

```
root@parkhk-redhat9:~  
[root@ez-x5 nfs]$ ./loop_unrolling  
1 loop unrolling 0sec 1183000nsec  
2 loop unrolling 0sec 628000nsec  
4 loop unrolling 0sec 600000nsec  
8 loop unrolling 0sec 575000nsec  
16 loop unrolling 0sec 574000nsec  
[root@ez-x5 nfs]$
```

- unrolling 개수에 비례하여 실행시간이 줄어 들지는 않음

Loop Unrolling

- unrolling 개수와 최적화 그리고 프로그램 사이즈의 관계

✓ 파일 크기

```
root@parkhk-redhat9:~  
[root@ez-x5 nfs]$ ls -l  
-rwxr-xr-x  1 root  root    13951 Nov  5  2007 loop_unrolling  
-rwxr-xr-x  1 root  root    11967 Nov  5  2007 loop_unrolling1  
-rwxr-xr-x  1 root  root    12513 Nov  5  2007 loop_unrolling16  
-rwxr-xr-x  1 root  root    12063 Nov  5  2007 loop_unrolling4  
-rwxr-xr-x  1 root  root    12223 Nov  5  2007 loop_unrolling8  
[root@ez-x5 nfs]$
```

- unrolling 개수에 비례하여 파일의 크기가 커짐

Loop Unrolling

■ Loop Unrolling

✓ 수행시간

- unrolling 개수에 비례하지 않음
- register이용 개수를 고려하여 작성해야 최적화의 의미가 있음
 - unrolling을 많이 할수록 필요한 register의 개수가 많아짐

✓ 메모리 사용률

- Unrolling의 깊이가 증가 할수록 메모리 사용량 증가

Loop Invariant Code Motion

- Loop Invariant Code Motion
 - ✓ Loop 최적화 기법
 - ✓ Loop내에 불변하는 계산들은 오버헤드를 유발함
 - ✓ Loop내에서 불변하는 계산 값이나 함수를 코드 바깥쪽으로 분리
 - Example
 - strlen()이 비교문에 들어가는 경우
 - 변하지 않는 사칙 연산 값
 - ✓ loop내에 불변의 값을 상수 값으로 대체

Loop Invariant Code Motion

- Loop Invariant Code Motion

예제코드

```
/* original source code (calculation)*/  
  
int array_v[ARRAY_SIZE];  
  
int loop_ICM1(int v1, int v2, int v3){  
  
    int i;  
  
    for(i=0; i<ARRAY_SIZE ; i++){  
        array_v[i] = ((v1*v1)+(v2*v2)+(v3*v3)+ARRAY_SIZE);  
    }  
  
}
```

Loop Invariant Code Motion

- Loop Invariant Code Motion

예제코드

```
/*loop ICM technique (calculation) */  
  
int array_v[ARRAY_SIZE];  
  
int loop_ICM2(int v1, int v2, int v3){  
  
    int i;  
    int tmp;  
  
    tmp = ((v1*v1)+(v2*v2)+(v3*v3)+ARRAY_SIZE);  
  
    for(i=0; i<ARRAY_SIZE ; i++){  
        array_v[i] = tmp;  
    }  
}
```


Loop Invariant Code Motion

■ 예제 코드 비교

✓ loop_ICM1()

- ARRAY_SIZE횟수 loop를 수행하면서 매번 매개변수들의 제곱에 ARRAY_SIZE를 더하여 배열을 초기화 $((v1*v1)+(v2*v2)+(v3*v3)+ARRAY_SIZE)$ 를 계산하여 배열에 저장

✓ loop_ICM2()

- 매개변수들의 제곱에 ARRAY_SIZE를 loop위에서 미리 계산하여 loop내에서 계산과정을 생략

✓ 불필요한 계산과정을 생략 함으로써 코드를 최적화

Loop Invariant Code Motion

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o loop_ICM loop_ICM.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_ICM
116
117 0000847c <loop_ICM1>:
118      847c:      e1a0c00d      mov     ip, sp
119      8480:      e92dd800      stmdb  sp!, {fp, ip, lr, pc}
120      8484:      e24cb004      sub    fp, ip, #4      ; 0x4
121      8488:      e24dd010      sub    sp, sp, #16     ; 0x10
122      848c:      e50b0010      str    r0, [fp, -#16]
123      8490:      e50b1014      str    r1, [fp, -#20]
124      8494:      e50b2018      str    r2, [fp, -#24]
125      8498:      e3a03000      mov    r3, #0      ; 0x0
126      849c:      e50b301c      str    r3, [fp, -#28]
127      84a0:      e51b201c      ldr    r2, [fp, -#28]
128      84a4:      e3a03a02      mov    r3, #8192     ; 0x2000
129      84a8:      e2833003      add    r3, r3, #3     ; 0x3
130      84ac:      e1520003      cmp    r2, r3
131      84b0:      da000000      ble   84b8 <loop_ICM1+0x3c>
132      84b4:      ea000013      b     8508 <loop_ICM1+0x8c>
```

함수명

argument를 stack
에 저장하고 for문
시작 부분

116,0-1 318

Loop Invariant Code Motion

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o loop_ICM loop_ICM.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_ICM
133 84b8: e59f0050 ldr r0, [pc, #80] ; 8510 <loop_ICM1+0x94>
134 84bc: e51bc01c ldr ip, [fp, #-28]
135 84c0: e51b2010 ldr r2, [fp, #-16]
136 84c4: e51b3010 ldr r3, [fp, #-16]
137 84c8: e0010293 mul r1, r3, r2
138 84cc: e51b2014 ldr r2, [fp, #-20]
139 84d0: e51b3014 ldr r3, [fp, #-20]
140 84d4: e0030392 mul r3, r2, r3
141 84d8: e0811003 add r1, r1, r3
142 84dc: e51b2018 ldr r2, [fp, #-24]
143 84e0: e51b3018 ldr r3, [fp, #-24]
144 84e4: e0030392 mul r3, r2, r3
145 84e8: e0813003 add r3, r1, r3
146 84ec: e2833a02 add r3, r3, #8192 ; 0x2000
147 84f0: e2833004 add r3, r3, #4 ; 0x4
148 84f4: e780310c str r3, [r0, ip, lsl #2]
149 84f8: e51b301c ldr r3, [fp, #-28]
150 84fc: e2833001 add r3, r3, #1 ; 0x1
151 8500: e50b301c str r3, [fp, #-28]
152 8504: eaffffe5 b 84a0 <loop_ICM1+0x24>
153 8508: e1a00003 mov r0, r3
154 850c: e91ba800 ldmdb fp, {fp, sp, pc}
155 8510: 00010a00 andeq r0, r1, r0, lsl #20
156
```

함수명:
loop_ICM1()

loop안에서 복
잡한 계산과정
을 거쳐 배열에
값을 저장함

133,5 368

Loop Invariant Code Motion

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o loop_ICM loop_ICM.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_ICM
156
157 00008514 <loop_ICM2>:
158 8514: e1a0c00d mov ip, sp
159 8518: e92dd800 stmdb sp!, {fp, ip, lr, pc}
160 851c: e24cb004 sub fp, ip, #4 ; 0x4
161 8520: e24dd014 sub sp, sp, #20 ; 0x14
162 8524: e50b0010 str r0, [fp, #-16]
163 8528: e50b1014 str r1, [fp, #-20]
164 852c: e50b2018 str r2, [fp, #-24]
165 8530: e51b2010 ldr r2, [fp, #-16]
166 8534: e51b3010 ldr r3, [fp, #-16]
167 8538: e0010293 mul r1, r3, r2
168 853c: e51b2014 ldr r2, [fp, #-20]
169 8540: e51b3014 ldr r3, [fp, #-20]
170 8544: e0030392 mul r3, r2, r3
171 8548: e0811003 add r1, r1, r3
172 854c: e51b2018 ldr r2, [fp, #-24]
173 8550: e51b3018 ldr r3, [fp, #-24]
174 8554: e0030392 mul r3, r2, r3
175 8558: e0813003 add r3, r1, r3
176 855c: e2833a02 add r3, r3, #8192 ; 0x2000
177 8560: e2833004 add r3, r3, #4 ; 0x4
178 8564: e50b3020 str r3, [fp, #-32]
179 8568: e3a03000 mov r3, #0 ; 0x0
180 856c: e50b301c str r3, [fp, #-28]
181 8570: e51b201c ldr r2, [fp, #-28]
182 8574: e3a03a02 mov r3, #8192 ; 0x2000
183 8578: e2833003 add r3, r3, #3 ; 0x3
184 857c: e1520003 cmp r2, r3
185 8580: da000000 ble 8588 <loop_ICM2+0x74>
186 8584: ea000007 b 85a8 <loop_ICM2+0x94>
```

함수명

loop 밖에서 미리 복잡한 계산을 하여 stack에 저장

Loop Invariant Code Motion

- O0 옵션을 이용한 컴파일 결과

- ✓ `armv5l-linux-gcc -O0 -lrt -o loop_ICM loop_ICM.c`

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_ICM
187  8588:    e59f1020    ldr    r1, [pc, #32]    ; 85b0 <loop_ICM2+0x9c>
188  858c:    e51b201c    ldr    r2, [fp, #-28]
189  8590:    e51b3020    ldr    r3, [fp, #-32]
190  8594:    e7813102    str    r3, [r1, r2, lsl #2]
191  8598:    e51b301c    ldr    r3, [fp, #-28]
192  859c:    e2833001    add   r3, r3, #1      ; 0x1
193  85a0:    e50b301c    str    r3, [fp, #-28]
194  85a4:    eafffff1    b     8570 <loop_ICM2+0x5c>
195  85a8:    e1a00003    mov   r0, r3
196  85ac:    e91ba800    ldmdb fp, {fp, sp, pc}
197  85b0:    00010a00    andeq r0, r1, r0, lsl #20
```



loop안에서 미리 계산된 값을 배열에 저장

함수명: loop_ICM2()

187,5 498

Loop Invariant Code Motion

■ O0 옵션을 이용한 컴파일 결과

- ✓  부분은 for문을 수행하기 위한 arm assembly이고  부분은 수식을 계산하여 배열에 저장하는 부분.

✓ loop_ICM1()

- 수식을 계산하여 배열에 저장하기 위한 명령어 개수
 - ldr : 8회
 - mul : 3회
 - add : 4회
 - str : 1회
- 위의 명령어들이 ARRAY_SIZE만큼 반복

✓ loop_ICM2()

- 수식을 계산하여 배열에 저장하기 위한 명령어 개수
 - for문 밖에서 계산되어지는 부분
 - ldr : 8
 - mul : 3
 - add : 4
 - str : 1
 - for문 안에서 계산되어지는 부분
 - ldr : 3
 - str : 1

Loop Invariant Code Motion

- OO옵션을 이용한 컴파일 결과
 - ✓ 수행 결과(ez-x5)
 - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 876000nsec, 645000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_loop_ICM]$ ./loop_ICM  
  
loop_ICM1 ()  
Iteration : 8196  
0sec 876000nsec  
  
loop_ICM2 ()  
Iteration : 8196  
0sec 645000nsec  
[root@ez-x5 dir_loop_ICM]$
```

Loop Invariant Code Motion

■ O3 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O3 -lrt -o loop_ICM loop_ICM.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_ICM
000085e4 <loop_ICM1>:
85e4: e00c0191 mul ip, r1, r1
85e8: e021c090 mla r1, r0, r0, ip
85ec: e59f0028 ldr r0, [pc, #40] ; 861c <loop_ICM1+0x38>
85f0: e02c1292 mla ip, r2, r2, r1
85f4: e3a02000 mov r2, #0 ; 0x0
85f8: e28c1a02 add r1, ip, #8192 ; 0x2000
85fc: e3a0ca02 mov ip, #8192 ; 0x2000
8600: e2813004 add r3, r1, #4 ; 0x4
8604: e28c1003 add r1, ip, #3 ; 0x3
8608: e7803102 str r3, [r0, r2, lsl #2]
860c: e2822001 add r2, r2, #1 ; 0x1
8610: e1520001 cmp r2, r1
8614: daffffff ble 8608 <loop_ICM1+0x24>
8618: e12fff1e bx lr
861c: 000108a0 andeq r0, r1, r0, lsr #17
:
00008620 <loop_ICM2>:
8620: e00c0191 mul ip, r1, r1
8624: e021c090 mla r1, r0, r0, ip
8628: e59f0028 ldr r0, [pc, #40] ; 8658 <loop_ICM2+0x38>
862c: e02c1292 mla ip, r2, r2, r1
8630: e3a02000 mov r2, #0 ; 0x0
8634: e28c1a02 add r1, ip, #8192 ; 0x2000
8638: e3a0ca02 mov ip, #8192 ; 0x2000
863c: e2813004 add r3, r1, #4 ; 0x4
8640: e28c1003 add r1, ip, #3 ; 0x3
8644: e7803102 str r3, [r0, r2, lsl #2]
8648: e2822001 add r2, r2, #1 ; 0x1
864c: e1520001 cmp r2, r1
8650: daffffff ble 8644 <loop_ICM2+0x24>
8654: e12fff1e bx lr
8658: 000108a0 andeq r0, r1, r0, lsr #17
:
200,1 788
```


Loop Invariant Code Motion

■ O3옵션을 이용한 컴파일 결과

- ✓ 컴파일러에 의한 최적화에 의해 같은 어셈블리어를 생성해냄
- ✓ 결과적으로 최적화 코드가 명시적인 변수 선언에 의해 약간의 오버헤드가 생김
- ✓ 수행 결과(ez-x5)
 - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 413000nsec, 453000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_loop_ICM]$ ./loop_ICM  
  
loop_ICM1 ()  
Iteration : 8196  
0sec 413000nsec  
  
loop_ICM2 ()  
Iteration : 8196  
0sec 453000nsec  
[root@ez-x5 dir_loop_ICM]$
```

Loop Invariant Code Motion

- Loop Invariant Code Motion

예제코드

```
/* original source code (function call) */  
  
int array_v[ARRAY_SIZE];  
  
int loop_ICM3(char *count){  
  
    int i;  
  
    for(i=0; i < atoi(count) ; i++){  
        array_v[i] = i;  
    }  
  
}
```

Loop Invariant Code Motion

- Loop Invariant Code Motion

예제코드

```
/*loop ICM technique (function call)*/  
  
int array_v[ARRAY_SIZE];  
  
int loop_ICM4(char *count){  
  
    int i;  
    int cnt;  
  
    cnt = atoi(count);  
    for(i=0; i<cnt ; i++){  
        array_v[i] = i;  
    }  
}
```

Loop Invariant Code Motion

■ 예제 코드 비교

- ✓ 매개변수로 전달 받은 문자를 정수형으로 변환하여 해당 숫자만큼 반복문을 수행함.
- ✓ loop_ICM3
 - 반복문 수행 시 매번 atoi()를 호출
- ✓ loop_ICM4
 - atoi()호출 결과 값을 상수 값으로 대체하여 loop내에서 호출 횟수를 줄임
- ✓ 호출 결과 값을 미리 저장해 놓음으로써 함수 호출 횟수를 줄임

Loop Invariant Code Motion

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o loop_ICM loop_ICM.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_ICM
000085b4 <loop_ICM3>:
85b4: e1a0c00d    mov     ip, sp
85b8: e92dd800    stmdb  sp!, {fp, ip, lr, pc}
85bc: e24cb004    sub    fp, ip, #4      ; 0x4
85c0: e24dd008    sub    sp, sp, #8     ; 0x8
85c4: e50b0010    str    r0, [fp, #-16]
85c8: e3a03000    mov    r3, #0        ; 0x0
85cc: e50b3014    str    r3, [fp, #-20]
85d0: e51b0010    ldr    r0, [fp, #-16]
85d4: ebffff60    bl     835c <init+0x54>
85d8: e1a02000    mov    r2, r0
85dc: e51b3014    ldr    r3, [fp, #-20]
85e0: e1530002    cmp    r3, r2
85e4: ba000000    blt   85ec <loop_ICM3+0x38>
85e8: ea000007    b     860c <loop_ICM3+0x58>
85ec: e59f1020    ldr    r1, [pc, #32]  ; 8614 <loop_ICM3+0x60>
85f0: e51b2014    ldr    r2, [fp, #-20]
85f4: e51b3014    ldr    r3, [fp, #-20]
85f8: e7813102    str    r3, [r1, r2, lsl #2]
85fc: e51b3014    ldr    r3, [fp, #-20]
8600: e2833001    add    r3, r3, #1     ; 0x1
8604: e50b3014    str    r3, [fp, #-20]
8608: eaafffff0    b     85d0 <loop_ICM3+0x1c>
860c: e1a00003    mov    r0, r3
8610: e91ba800    ldmdb fp, {fp, sp, pc}
8614: 00010a00    andeq r0, r1, r0, lsl #20
```

함수명

반복문 내에 함수 호출 구문이 있음. 반복 횟수만큼 호출됨

198,0-1 558

Loop Invariant Code Motion

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o loop_ICM loop_ICM.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_ICM
00008618 <loop_ICM4>:
8618: e1a0c00d      mov     ip, sp
861c: e92dd800      stmdb  sp!, {fp, ip, lr, pc}
8620: e24cb004      sub    fp, ip, #4      ; 0x4
8624: e24dd00c      sub    sp, sp, #12     ; 0xc
8628: e50b0010      str    r0, [fp, -#16]
862c: e51b0010      ldr    r0, [fp, -#16]
8630: ebffff49      bl     835c <init+0x54>
8634: e1a03000      mov    r3, r0
8638: e50b3018      str    r3, [fp, -#24]
863c: e3a03000      mov    r3, #0         ; 0x0
8640: e50b3014      str    r3, [fp, -#20]
8644: e51b2014      ldr    r2, [fp, -#20]
8648: e51b3018      ldr    r3, [fp, -#24]
864c: e1520003      cmp    r2, r3
8650: ba000000      blt   8658 <loop_ICM4+0x40>
8654: ea000007      b     8678 <loop_ICM4+0x60>
8658: e59f1020      ldr    r1, [pc, #32]   ; 8680 <loop_ICM4+0x68>
865c: e51b2014      ldr    r2, [fp, -#20]
8660: e51b3014      ldr    r3, [fp, -#20]
8664: e7813102      str    r3, [r1, r2, lsl #2]
8668: e51b3014      ldr    r3, [fp, -#20]
866c: e2833001      add    r3, r3, #1     ; 0x1
8670: e50b3014      str    r3, [fp, -#20]
8674: eafffff2      b     8644 <loop_ICM4+0x2c>
8678: e1a00003      mov    r0, r3
867c: e91ba800      ldmdb fp, {fp, sp, pc}
8680: 00010a00      andeq  r0, r1, r0, lsl #20
```

함수명

반복문 밖에 함수 호출 구문이 있음. 결과 값을 stack에 저장 해서 사용함

254,0-1 638

Loop Invariant Code Motion

■ OO옵션을 이용한 컴파일 결과

- ✓ ■부분은 for문에서 반복되는 구간이고 ■부분은 함수를 호출하는 부분이다
- ✓ loop_ICM3()
 - 반복문에서 루프를 수행할 때 매번 atoi()함수를 호출함.
- ✓ loop_ICM4()
 - 반복문에서 밖에서 atoi()함수를 호출하고 atoi()의 반환 값을 stack에 저장. 반복문 내에서는 stack에 있는 값을 사용.

Loop Invariant Code Motion

- OO옵션을 이용한 컴파일 결과

- ✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 14634000nsec, 658000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_loop_ICM]$ ./loop_ICM  
  
loop_ICM3 ()  
Iteration : 8196  
0sec 14634000nsec  
  
loop_ICM4 ()  
Iteration : 8196  
0sec 658000nsec  
[root@ez-x5 dir_loop_ICM]$
```


Loop Invariant Code Motion

■ O3 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O3 -lrt -o loop_ICM loop_ICM.c

```
root@parkhk-redhat9:~/embedded_board/optimize/loop_ICM

00008668 <loop_ICM3>:
8668: e92d4070 stmdb sp!, {r4, r5, r6, lr}
866c: e59f6020 ldr r6, [pc, #32] ; 8694 <loop_ICM3+0x2c>
8670: e1a05000 mov r5, r0
8674: e3a04000 mov r4, #0 ; 0x0
8678: e1a00005 mov r0, r5
867c: ebffff43 bl 8390 <_init+0x64>
8680: e1540000 cmp r4, r0
8684: b7864104 strlt r4, [r6, r4, lsl #2]
8688: b2844001 addlt r4, r4, #1 ; 0x1
868c: bafffff9 blt 8678 <loop_ICM3+0x10>
8690: e8bd8070 ldmia sp!, {r4, r5, r6, pc}
8694: 000108c0 andeq r0, r1, r0, asr #17

00008698 <loop_ICM4>:
8698: e52de004 str lr, [sp, -#4]!
869c: ebffff3b bl 8390 <_init+0x64>
86a0: e3a03000 mov r3, #0 ; 0x0
86a4: e1530000 cmp r3, r0
86a8: a49df004 ldrge pc, [sp], #4
86ac: e59f2010 ldr r2, [pc, #16] ; 86c4 <loop_ICM4+0x1c>
86b0: e7823103 str r3, [r2, r3, lsl #2]
86b4: e2833001 add r3, r3, #1 ; 0x1
86b8: e1530000 cmp r3, r0
86bc: bafffffb blt 86b0 <loop_ICM4+0x18>
86c0: e49df004 ldr pc, [sp], #4
86c4: 000108c0 andeq r0, r1, r0, asr #17
```

함수명

반복문 내에 함수 호출 구문이 있음. 반복 횟수만큼 호출됨

함수명

반복문 밖에 함수 호출 구문이 있음. 결과 값을 stack에 저장해서 사용함

264,0-1 908

Loop Invariant Code Motion

■ O3옵션을 이용한 컴파일 결과

- ✓ ■ 부분은 for문에서 반복되는 구간이고 ■ 부분은 함수를 호출하는 부분이다
- ✓ 수식을 계산하는 것과는 다르게 최적화 후에도 loop_ICM3에서 함수 호출되는 부분이 for문 안에 있음
- ✓ 수행 결과(ez-x5)
 - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 14532000nsec, 483000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_loop_ICM]$ ./loop_ICM  
  
loop_ICM3 ()  
Iteration : 8196  
0sec 14532000nsec  
  
loop_ICM4 ()  
Iteration : 8196  
0sec 483000nsec  
[root@ez-x5 dir_loop_ICM]$ █
```

Loop Invariant Code Motion

■ Loop Invariant Code Motion

- ✓ 불변하는 코드를 루프 밖으로 이동시켜 결과값을 stack에 저장하여 최적화를 할 수 있음.
 - 수식이 간단하다면?
 - ldr명령어는 메모리에 접근하므로 비용이 많이 듦. loop안에서 계산하는 것이 더 효율적 일 수 있음
 - ex) 계산 값이 define값을 이용하여 간단한 상수로 계산되어 지는 경우
- ✓ O3옵션을 이용하면 최적화를 해줌
- ✓ 함수 호출의 경우 O0에 대비하여 O3옵션으로 컴파일 하더라도 이득이 별로 없음
 - 함수를 호출하여 수행하기 때문

Strength Reduction

■ Strength Reduction

- ✓ 고비용의 연산자들 저비용 연산자를 이용하여 연산되도록 코드를 최적화하는 기법

- Example

- 나머지 연산을 비트 연산으로 대체
- `sin`, `cos`, `pow` 등 수학 함수를 호출하는 루틴 중 연산자로 계산시 간단하게 구현 가능한 부분에 대하여 연산자로 대체

Strength Reduction

- Strength Reduction

예제코드

```
/* original source code (Operator)*/  
  
int array_v[ARRAY_SIZE];  
  
int strength_reduction1(void){  
  
    int i;  
  
    for(i=0; i < ARRAY_SIZE ; i++){  
        array_v[i] = i % 8;  
    }  
}
```

Strength Reduction

- Strength Reduction

예제코드

```
/*Strength reduction technique (Operator) */  
  
int array_v[ARRAY_SIZE];  
  
int strength_reduction2(void){  
  
    int i;  
  
    for(i=0; i < ARRAY_SIZE ; i++){  
        array_v[i] = i & 7;  
    }  
}
```

Strength Reduction

■ 예제 코드 비교

✓ strength_reduction1

- 반복문 수행 시 매번 나머지 연산
- 나머지 연산은 많은 비용이 듦

✓ strength_reduction2

- 반복문 수행 시 나머지 연산을 비트 연산으로 대체
- 나머지 연산에 비하여 적은 비용으로 같은 결과 값을 산출함

Strength Reduction

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o strength_reduction strength_reduction.c

```
root@parkhk-redhat9:~/embedded_board/optimize/strength_reduction_dir
000084a8 <strength_reduction1>:
84a8: e1a0c00d mov ip, sp
84ac: e92dd800 stmdb sp!, {fp, ip, lr, pc}
84b0: e24cb004 sub fp, ip, #4 ; 0x4
84b4: e24dd004 sub sp, sp, #4 ; 0x4
84b8: e3a03000 mov r3, #0 ; 0x0
84bc: e50b3010 str r3, [fp, #-16]
84c0: e51b2010 ldr r2, [fp, #-16]
84c4: e3a03d7e mov r3, #8064 ; 0x1f80
84c8: e283301b add r3, r3, #27 ; 0x1b
84cc: e1520003 cmp r2, r3
84d0: da000000 ble 84d8 <strength_reduction1+0x30>
84d4: ea00000d b 8510 <strength_reduction1+0x68>
84d8: e59f1038 ldr r1, [pc, #56] ; 8518 <strength_reduction1+0x70>
84dc: e51b0010 ldr r0, [fp, #-16]
84e0: e51b2010 ldr r2, [fp, #-16]
84e4: e1a03fc2 mov r3, r2, asr #31
84e8: e1a03ea3 mov r3, r3, lsr #29
84ec: e0823003 add r3, r2, r3
84f0: e1a031c3 mov r3, r3, asr #3
84f4: e1a03183 mov r3, r3, lsl #3
84f8: e0633002 rsb r3, r3, r2
84fc: e7813100 str r3, [r1, r0, lsl #2]
8500: e51b3010 ldr r3, [fp, #-16]
8504: e2833001 add r3, r3, #1 ; 0x1
8508: e50b3010 str r3, [fp, #-16]
850c: eaffffeb b 84c0 <strength_reduction1+0x18>
8510: e1a00003 mov r0, r3
8514: e91ba800 ldmdb fp, {fp, sp, pc}
8518: 000109e0 andeq r0, r1, r0, ror #19
```

함수명

나머지 연산을 위한 assembly

147,0-1 358

Strength Reduction

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o strength_reduction strength_reduction.c

```
root@parkhk-redhat9:~/embedded_board/optimize/strength_reduction_dir
0000851c <strength_reduction2>
851c: e1a0c00d mov ip, sp
8520: e92dd800 stmdb sp!, {fp, ip, lr, pc}
8524: e24cb004 sub fp, ip, #4 ; 0x4
8528: e24dd004 sub sp, sp, #4 ; 0x4
852c: e3a03000 mov r3, #0 ; 0x0
8530: e50b3010 str r3, [fp, -#16]
8534: e51b2010 ldr r2, [fp, -#16]
8538: e3a03d7e mov r3, #8064 ; 0x1f80
853c: e283301b add r3, r3, #27 ; 0x1b
8540: e1520003 cmp r2, r3
8544: da000000 ble 854c <strength_reduction2+0x30>
8548: ea000008 b 8570 <strength_reduction2+0x54>
854c: e59f1024 ldr r1, [pc, #36] ; 8578 <strength_reduction2+0x5c>
8550: e51b2010 ldr r2, [fp, -#16]
8554: e51b3010 ldr r3, [fp, -#16]
8558: e2033007 and r3, r3, #7 ; 0x7
855c: e7813102 str r3, [r1, r2, lsl #2]
8560: e51b3010 ldr r3, [fp, -#16]
8564: e2833001 add r3, r3, #1 ; 0x1
8568: e50b3010 str r3, [fp, -#16]
856c: eafffff0 b 8534 <strength_reduction2+0x18>
8570: e1a00003 mov r0, r3
8574: e91ba800 ldmdb fp, {fp, sp, pc}
8578: 000109e0 andeq r0, r1, r0, ror #19
```

Strength Reduction

- O0 옵션을 이용한 컴파일 결과

- ✓  부분이 나머지 연산을 하기 위한 assembly 코드

- ✓ strength_reduction1()

- % 연산 비용

- ldr : 3회
- mov : 4회
- add : 1회
- rsb : 1회
- str : 1회

- ✓ strength_reduction2()

- % 연산 비용

- ldr : 3회
- and : 1회
- str : 1회

Strength Reduction

- OO옵션을 이용한 컴파일 결과
 - ✓ 수행 결과(ez-x5)
 - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 707000nsec, 615000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_strength_reduction]$ ./strength_reduction  
  
strength_reduction1()  
Iteration : 8092  
0sec 707000nsec  
  
strength_reduction2()  
Iteration : 8092  
0sec 615000nsec  
[root@ez-x5 dir_strength_reduction]$ █
```

Strength Reduction

■ O3 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O3 -lrt -o strength_reduction strength_reduction.c

```
root@parkhk-redhat9:~/embedded_board/optimize/strength_reduction_dir
00008618 <strength_reduction1>:
8618: e59f002c ldr r0, [pc, #44] ; 864c <strength_reduction1+0x34>
861c: e3a03d7e mov r3, #8064 ; 0x1f80
8620: e3a02000 mov r2, #0 ; 0x0
8624: e283101b add r1, r3, #27 ; 0x1b
8628: e1a03fc2 mov r3, r2, asr #31
862c: e082cea3 add ip, r2, r3, lsr #29
8630: e3cc3007 bic r3, ip, #7 ; 0x7
8634: e063c002 rsb ip, r3, r2
8638: e780c102 str ip, [r0, r2, lsl #2]
863c: e2822001 add r2, r2, #1 ; 0x1
8640: e1520001 cmp r2, r1
8644: dafffff7 ble 8628 <strength_reduction1+0x10>
8648: e12fff1e bx lr
864c: 00010900 andeq r0, r1, r0, lsl #18

00008650 <strength_reduction2>:
8650: e59f0020 ldr r0, [pc, #32] ; 8678 <strength_reduction2+0x28>
8654: e3a03d7e mov r3, #8064 ; 0x1f80
8658: e3a02000 mov r2, #0 ; 0x0
865c: e283101b add r1, r3, #27 ; 0x1b
8660: e202c007 and ip, r2, #7 ; 0x7
8664: e780c102 str ip, [r0, r2, lsl #2]
8668: e2822001 add r2, r2, #1 ; 0x1
866c: e1520001 cmp r2, r1
8670: dafffffa ble 8660 <strength_reduction2+0x10>
8674: e12fff1e bx lr
8678: 00010900 andeq r0, r1, r0, lsl #18
```

함수명

나머지 연산을 위한 assembly

함수명

비트 연산을 위한 assembly

202,0-1 768

Strength Reduction

■ O3 옵션을 이용한 컴파일 결과

✓  부분이 나머지 연산을 하기 위한 assembly 코드

✓ strength_reduction1()

▪ % 연산 비용

- mov : 1회
- add : 1회
- rsb : 1회
- bic : 1회
- str : 1회

✓ strength_reduction2()

▪ % 연산 비용

- and : 1회
- str : 1회

✓ 최적화 옵션으로 O3을 사용하였을 경우 고비용의 연산자가 없고 코드가 간결해짐

✓ 실험 결과 거의 성능차가 거의 없음

Strength Reduction

■ O3옵션을 이용한 컴파일 결과

✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 470000nsec, 468000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_strength_reduction]$ ./strength_reduction  
  
strength_reduction1()  
Iteration : 8092  
0sec 470000nsec  
  
strength_reduction2()  
Iteration : 8092  
0sec 468000nsec  
[root@ez-x5 dir_strength_reduction]$
```

Strength Reduction

- Strength Reduction

예제코드

```
/* original source code (Function call)*/  
int strength_reduction3(void){  
    int i;  
  
    for(i=0; i < ARRAY_SIZE ; i++){  
        array_v[i] = pow(i, 2);  
    }  
}
```

Strength Reduction

- Strength Reduction

예제코드

```
/*Strength reduction technique (Function call) */  
int strength_reduction4(void){  
    int i;  
  
    for(i=0; i < ARRAY_SIZE ; i++){  
        array_v[i] = i*i;  
    }  
}
```


Strength Reduction

■ 예제 코드 비교

✓ strength_reduction3

- 반복문 수행 시 매번 `pow()` 함수 호출하여 제곱 값 산출
- 수학 함수를 호출하여 값 산출 시 많은 비용이 듦

✓ strength_reduction3

- 반복문 수행 시 수학 함수 호출을 산술연산으로 대체
- 수학 함수 호출에 비하여 적은 비용으로 같은 결과 값을 산출함

Strength Reduction

■ 00옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o strength_reduction strength_reduction.c

```
root@parkhk-redhat9:~/embedded_board/optimize/strength_reduction_dir
0000857c <strength_reduction3>:
857c: e1a0c00d mov ip, sp
8580: e92dd830 stmdb sp!, {r4, r5, fp, ip, lr, pc}
8584: e24cb004 sub fp, ip, #4 ; 0x4
8588: e24dd004 sub sp, sp, #4 ; 0x4
858c: e3a03000 mov r3, #0 ; 0x0
8590: e50b3018 str r3, [fp, -#24]
8594: e51b2018 ldr r2, [fp, -#24]
8598: e3a03d7e mov r3, #8064 ; 0x1f80
859c: e283301b add r3, r3, #27 ; 0x1b
85a0: e1520003 cmp r2, r3
85a4: da000000 ble 85ac <strength_reduction3+0x30>
85a8: ea00000e b 85e8 <strength_reduction3+0x6c>
85ac: e59f4044 ldr r4, [pc, #68] ; 85f8 <strength_reduction3+0x7c>
85b0: e51b5018 ldr r5, [fp, -#24]
85b4: e51b3018 ldr r3, [fp, -#24]
85b8: ee003190 fliTd f0, r3
85bc: ed2d8102 stfd f0, [sp, -#8]!
85c0: e8bd0003 ldmia sp!, {r0, r1}
85c4: e28f2024 add r2, pc, #36 ; 0x24
85c8: e892000c ldmia r2, {r2, r3}
85cc: ebffff61 bl 8358 <_init+0x24>
85d0: ee103170 fixz r3, f0
85d4: e7843105 str r3, [r4, r5, lsl #2]
85d8: e51b3018 ldr r3, [fp, -#24]
85dc: e2833001 add r3, r3, #1 ; 0x1
85e0: e50b3018 str r3, [fp, -#24]
85e4: eaaffffea b 8594 <strength_reduction3+0x18>
85e8: e1a00003 mov r0, r3
85ec: e91ba830 ldmdb fp, {r4, r5, fp, sp, pc}
85f0: 40000000 andmi r0, r0, r0
85f4: 00000000 andeq r0, r0, r0
85f8: 000109e0 andeq r0, r1, r0, ror #19
```

함수명

지수승을 구하기 위해 pow함수 호출. 반환값을 배열에 저장하는 assembly code

207,0-1 538

Strength Reduction

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o strength_reduction strength_reduction.c


```
root@parkhk-redhat9:~/embedded_board/optimize/strength_reduction_dir
000085fc <strength_reduction4>:
85fc:    e1a0c00d    mov     ip, sp
8600:    e92dd800    stmdb  sp!, {fp, ip, lr, pc}
8604:    e24cb004    sub    fp, ip, #4      ; 0x4
8608:    e24dd004    sub    sp, sp, #4     ; 0x4
860c:    e3a03000    mov    r3, #0        ; 0x0
8610:    e50b3010    str    r3, [fp, #-16]
8614:    e51b2010    ldr    r2, [fp, #-16]
8618:    e3a03d7e    mov    r3, #8064     ; 0x1f80
861c:    e283301b    add    r3, r3, #27   ; 0x1b
8620:    e1520003    cmp    r2, r3
8624:    da000000    ble   862c <strength_reduction4+0x30>
8628:    ea000009    b     8654 <strength_reduction4+0x58>
862c:    e59f0028    ldr    r0, [pc, #40] ; 865c <strength_reduction4+0x60>
8630:    e51b1010    ldr    r1, [fp, #-16]
8634:    e51b2010    ldr    r2, [fp, #-16]
8638:    e51b3010    ldr    r3, [fp, #-16]
863c:    e0030392    mul    r3, r2, r3
8640:    e7803101    str    r3, [r0, r1, lsl #2]
8644:    e51b3010    ldr    r3, [fp, #-16]
8648:    e2833001    add    r3, r3, #1    ; 0x1
864c:    e50b3010    str    r3, [fp, #-16]
8650:    eaffffef    b     8614 <strength_reduction4+0x18>
8654:    e1a00003    mov    r0, r3
8658:    e91ba800    ldmdb fp, {fp, sp, pc}
865c:    000109e0    andeq  r0, r1, r0, ror #19
```

233,5

628

Strength Reduction

■ O0 옵션을 이용한 컴파일 결과

✓  부분이 제공 연산을 하기 위한 assembly 코드

✓ strength_reduction3()

▪ 제공 연산 비용(pow())호출)

- ldr : 3회
- fltd : 1회
- stfd : 1회
- ldmia : 2회
- add : 1회
- fixz : 1회
- str : 1회
- bl : 1회

✓ strength_reduction4()

▪ 제공 연산 비용(산술연산)

- ldr : 4회
- mul : 1회
- str : 1회

Strength Reduction

- OO옵션을 이용한 컴파일 결과
 - ✓ 수행 결과(ez-x5)
 - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 259531000nsec, 678000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_strength_reduction]$ ./strength_reduction  
  
strength_reduction3()  
Iteration : 8092  
0sec 259531000nsec  
  
strength_reduction4()  
Iteration : 8092  
0sec 678000nsec  
[root@ez-x5 dir_strength_reduction]$ █
```

Strength Reduction

■ O3 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O3 -lrt -o strength_reduction strength_reduction.c

```
root@parkhk-redhat9:~/embedded_board/optimize/strength_reduction_dir
00008688 <strength_reduction3>:
8688: e92d4070 stmdb sp!, {r4, r5, r6, lr}
868c: e59f6040 ldr r6, [pc, #64] ; 86d4 <strength_reduction3+0x4c>
8690: e3a00d7e mov r0, #8064 ; 0x1f80
8694: e3a04000 mov r4, #0 ; 0x0
8698: e280501b add r5, r0, #27 ; 0x1b
869c: ee014190 fldt f1, r4
86a0: e28f2024 add r2, pc, #36 ; 0x24
86a4: e892000c ldmbia r2, {r2, r3}
86a8: ed2d9102 stfd f1, [sp, #-8]!
86ac: e8bd0003 ldmbia sp!, {r0, r1}
86b0: ebffff30 bl 8378 <_init+0x24>
86b4: ee100170 fixz r0, f0
86b8: e7860104 str r0, [r6, r4, lsl #2]
86bc: e2844001 add r4, r4, #1 ; 0x1
86c0: e1540005 cmp r4, r5
86c4: dafffff4 ble 869c <strength_reduction3+0x14>
86c8: e8bd8070 ldmbia sp!, {r4, r5, r6, pc}
86cc: 40000000 andmi r0, r0, r0
86d0: 00000000 andeq r0, r0, r0
86d4: 00010900 andeq r0, r1, r0, lsl #18

000086d8 <strength_reduction4>:
86d8: e59f0020 ldr r0, [pc, #32] ; 8700 <strength_reduction4+0x28>
86dc: e3a03d7e mov r3, #8064 ; 0x1f80
86e0: e3a02000 mov r2, #0 ; 0x0
86e4: e283101b add r1, r3, #27 ; 0x1b
86e8: e00c0292 mul ip, r2, r2
86ec: e780c102 str ip, [r0, r2, lsl #2]
86f0: e2822001 add r2, r2, #1 ; 0x1
86f4: e1520001 cmp r2, r1
86f8: dafffffa ble 86e8 <strength_reduction4+0x10>
86fc: e12fff1e bx lr
8700: 00010900 andeq r0, r1, r0, lsl #18
```

함수명


지수승을 구하기 위해 pow 함수 호출. 반환값을 배열에 저장하는 assembly code

함수명

제곱을 구하기 위해 산술 연산을 하는 assembly code

Strength Reduction

■ O3 옵션을 이용한 컴파일 결과

- ✓  부분이 제공 연산을 하기 위한 assembly 코드
- ✓ strength_reduction3()
 - O0 옵션으로 컴파일 했을 경우와 비교하여 상대적인 성능 향상은 미비함
 - 함수 호출로 인한 오버헤드가 대부분
- ✓ strength_reduction3()
 - O0 옵션으로 컴파일 했을 경우와 비교하여 상대적으로 70% 정도의 성능 향상이 있음.

Strength Reduction

■ O3옵션을 이용한 컴파일 결과

✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 257165000nsec, 486000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_strength_reduction]$ ./strength_reduction  
  
strength_reduction3()  
Iteration : 8092  
0sec 257165000nsec  
  
strength_reduction4()  
Iteration : 8092  
0sec 486000nsec  
[root@ez-x5 dir_strength_reduction]$
```


Strength Reduction

■ Strength Reduction

- ✓ modular 연산 등 고비용의 산술연산을 비트 연산등의 저비용 연산으로 교체 하는 것이 효과적
- ✓ 수학 함수를 호출하여 연산을 하는 것은 산술연산에 비하여 고비용이 듦
- ✓ 간단한 연산의 경우 산술연산을 하는 것이 효과적
- ✓ 코드의 복잡도, 가독성, 속도 향상 등을 고려하여 최적화

Count up to Zero

■ Count up to Zero

- ✓ 루프의 종료 조건을 0과 비교하는 것이 효율적
- ✓ 대부분의 아키텍처에서는 0에 도달할 때 ZERO 플래그를 재설정하게 됨
- ✓ 변수를 0과 비교하게 되면 명시적인 비교문이 생략됨

Count up to Zero

- Count up to Zero

예제코드

```
/* original source code */  
  
int count_up_to_zero1(int *input){  
  
    int i;  
    int sum = 0;  
  
    for(i=0; i<ARRAY_SIZE ; ++i){  
        sum += *(input++)+i;  
    }  
  
    return sum;  
}
```

Count up to Zero

- Count up to Zero

예제코드

```
/*Count up to Zero*/  
  
int count_up_to_zero2(int *input){  
  
    int i;  
    int sum = 0;  
  
    for(i=ARRAY_SIZE-1; i!=0 ; --i){  
        sum += *(input++)+i;  
    }  
    sum += *(input++)+i;  
  
    return sum;  
}
```

Count up to Zero

■ 예제 코드 비교

✓ Count_up_to_zero1

- 루프문 종료 조건을 양의 상수와 비교
- 조건문에 사용되는 변수 `i`를 loop 내부에서 사용

✓ count_up_to_zero2

- 루프문 종료 조건을 0과 비교
- 조건문에 사용되는 변수 `i`를 loop 내부에서 사용

Count up to Zero

■ O3 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O3 -lrt -o count_up_to_zero count_up_to_zero2.c

```
0000860c <count_up_to_zero1>:
860c: e3a01000 mov r1, #0 ; 0x0
8610: e3a03a02 mov r3, #8192 ; 0x2000
8614: e1a02001 mov r2, r1
8618: e283c003 add ip, r3, #3 ; 0x3
861c: e4903004 ldr r3, [r0], #4
8620: e0833002 add r3, r3, r2
8624: e2822001 add r2, r2, #1 ; 0x1
8628: e152000c cmp r2, ip
862c: e0811003 add r1, r1, r3
8630: c1a00001 movgt r0, r1
8634: c12fff1e bxgt lr
8638: eafffff7 b 861c <count_up_to_zero1+0x10>

0000863c <count_up_to_zero2>:
863c: e3a03a02 mov r3, #8192 ; 0x2000
8640: e1a01000 mov r1, r0
8644: e3a0c000 mov ip, #0 ; 0x0
8648: e2832003 add r2, r3, #3 ; 0x3
864c: e4913004 ldr r3, [r1], #4
8650: e0833002 add r3, r3, r2
8654: e2522001 subs r2, r2, #1 ; 0x1
8658: e08cc003 add ip, ip, r3
865c: 1afffffa bne 864c <count_up_to_zero2+0x10>
8660: e3a02902 mov r2, #32768 ; 0x8000
8664: e282100c add r1, r2, #12 ; 0xc
8668: e7902001 ldr r2, [r0, r1]
866c: e08c1002 add r1, ip, r2
8670: e1a00001 mov r0, r1
8674: e12fff1e bx lr
```

함수명

반복 수행되는 부분. cmp 명령이 수행됨

함수명

반복 수행되는 부분. cmp 명령이 수행 않됨

"count_ut_zero_03" 271L, 10392C 215,0-1 898

Count up to Zero

■ O3옵션을 이용한 컴파일 결과

- ✓ 명시적인 조건 비교(cmp)를 하지 않음.
- ✓ 수행 결과(ez-x5)
 - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 513000nsec, 503000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_count_up_to_zero]$ ./count_ut_zero  
  
count_up_to_zero1()  
Iteration : 8196  
0sec 513000nsec  
Sum : 67166220  
  
count_up_to_zero2()  
Iteration : 8196  
0sec 503000nsec  
Sum : 67166220  
[root@ez-x5 dir_count_up_to_zero]$ █
```

Count up to Zero

- Count up to Zero

예제코드

```
/* original source code */  
  
int count_up_to_zero1(int *input){  
  
    int i;  
    int sum = 0;  
  
    for(i=0; i<ARRAY_SIZE ; ++i){  
        sum += *(input++);  
    }  
  
    return sum;  
}
```


Count up to Zero

- Count up to Zero

예제코드

```
/*Count up to Zero*/  
  
int count_up_to_zero2(int *input){  
  
    int i;  
    int sum = 0;  
  
    for(i=ARRAY_SIZE; i!=0 ; --i){  
        sum += *(input++);  
    }  
  
    return sum;  
}
```

Count up to Zero

■ 예제 코드 비교

✓ Count_up_to_zero1

- 루프문 종료 조건을 양의 상수와 비교
- 조건문에 사용되는 변수 `i`를 loop 내부에서 미사용

✓ count_up_to_zero2

- 루프문 종료 조건을 0과 비교
- 조건문에 사용되는 변수 `i`를 loop 내부에서 미사용

Count up to Zero

■ O3 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O3 -lrt -o count_up_to_zero count_up_to_zero2.c

```
root@parkhk-redhat9:~/embedded_board/optimize/count_up_to_zero
000085f4 <count_up_to_zero1>:
85f4: e3a03a02 mov r3, #8192 ; 0x2000
85f8: e3a01000 mov r1, #0 ; 0x0
85fc: e2832004 add r2, r3, #4 ; 0x4
8600: e490c004 ldr ip, [r0], #4
8604: e2522001 subs r2, r2, #1 ; 0x1
8608: e081100c add r1, r1, ip
860c: 01a00001 moveq r0, r1
8610: 012fff1e bxeq lr
8614: eafffff9 b 8600 <count_up_to_zero1+0xc>

00008618 <count_up_to_zero2>:
8618: e3a03a02 mov r3, #8192 ; 0x2000
861c: e3a01000 mov r1, #0 ; 0x0
8620: e2832004 add r2, r3, #4 ; 0x4
8624: e490c004 ldr ip, [r0], #4
8628: e2522001 subs r2, r2, #1 ; 0x1
862c: e081100c add r1, r1, ip
8630: 01a00001 moveq r0, r1
8634: 012fff1e bxeq lr
8638: eafffff9 b 8624 <count_up_to_zero2+0xc>
```

함수명

반복 수행되는 부분. cmp 명령이 수행 않됨

함수명

반복 수행되는 부분. cmp 명령이 수행 않됨

230,5 888

Count up to Zero

■ O3옵션을 이용한 컴파일 결과

- ✓ loop 내에서 조건 비교 변수(i)를 사용하지 않음으로써 최적화 되었음
- ✓ 결국 두 개의 예제는 똑같은 assembly code를 생성하였음
- ✓ 수행 결과(ez-x5)
 - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 463000nsec, 462000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_count_up_to_zero]$ ./count_ut_zero  
  
count_up_to_zero1()  
Iteration : 8196  
0sec 463000nsec  
Sum : 33583110  
  
count_up_to_zero2()  
Iteration : 8196  
0sec 462000nsec  
Sum : 33583110  
[root@ez-x5 dir_count_up_to_zero]$ █
```

Count up to Zero

- 조건 비교를 양수가 아닌 0값과 함으로써 명시적 `cmp` 명령이 생략 될 수 있음.
- 내부적으로 조건 비교 변수를 사용하지 않으면 최적화 옵션에 따라 `cmp` 명령이 없는 코드로 최적화 될 수 있음

Register Allocation

■ Register Allocation

✓ 레지스터 사용(일반적 용도)

- 컴파일러에서는 지역변수를 r3등에 저장하거나 sp를 ip에 저장하여 레지스터 사용을 최대화함

Reg. 번호	Reg. 이름	Reg. 사용
r0	a1	argument 1
r1	a2	argument 2
r2	a3	argument 3
r3	a4	argument 4
r4	v1	register variable 1
r5	v2	register variable 2
r6	v3	register variable 3
r7	v4	register variable 4

Reg. 번호	Reg. 이름	Reg. 사용
r8	v5	register variable 5
r9	v6/sb	register variable 6 / Static base
r10	v7/sl	register variable 7 /stack limit
r11(fp)	v8/fp	frame pointer
r12(ip)	ip	scratch register
r13(sp)	sp	stack pointer
r14(lr)	lr	link register
r15(pc)	pc	program counter

Register Allocation

■ Register Allocation

- ✓ C 컴파일러가 변수를 위해 사용할 수 있는 레지스터
 - r0~r12, r14
 - 실제로 몇몇 컴파일러에서는 리턴 주소를 저장하는 lr과 임시 작업용 레지스터인 ip는 할당하지 않음.
 - 레지스터 할당을 보장하기 위해서는 최대 12개의 register만을 사용하는 것이 바람직함
- ✓ 사용가능 register보다 많은 변수를 사용하면 stack에 저장
 - 변수들이 메모리에 저장됨(spilled)
- ✓ 최적화 방안
 - spilled되는 변수 최소화
 - 자주 접근되는 변수는 레지스터에 저장
 - 컴파일러의 register 할당에 의존하는 것이 바람직함

Register Allocation

■ Register Allocation

예제코드

```
/* Register Allocation*/  
  
int global_sum, global_a, global_s, global_m, global_d ;  
  
int reg_Alloc1(int v1, int v2){  
  
    global_a = v1 + v2;    global_s = v1 - v2;  
    global_m = v1 * v2;    global_d = v1 / v2;  
  
    //중간 생략 : 전역 변수를 이용하여 복잡한 연산 수행  
  
    global_sum = global_a + global_s + global_m + global_d;  
    global_sum &= 0xff00;  
    if(global_sum > 0) return 0;  
    return -1;  
}
```


Register Allocation

■ Register Allocation

예제코드

```
/* Register Allocation*/  
  
int global_sum, global_a, global_s, global_m, global_d ;  
  
int reg_Alloc2(int v1, int v2){  
  
    register int local_a;    register int local_s;    register int local_d;  
    register int local_m;    register int local_sum = 0;  
  
    //중간 생략 : 지역 변수를 이용하여 복잡한 연산 수행  
    //          (reg_Alloc1과 같은 연산 수행)  
  
    global_sum = local_sum;    global_a = local_a;    global_s = local_s;  
    global_m = local_m;    global_d = local_d;  
  
    if(local_sum > 0) return 0;  
  
    return -1;  
}
```

Register Allocation

■ 예제 코드 비교

✓ reg_Alloc1()

- 전역변수를 사용하여 복잡한 연산을 수행함

✓ reg_Alloc2()

- reg_Alloc1()과 수행 내용은 같음.
- 레지스터 변수를 선언하고 전역 변수의 값을 저장함
- 레지스터 변수를 사용하여 복잡한 연산을 수행
- 레지스터 변수의 값을 전역 변수에 저장

Register Allocation

- O0 옵션을 이용한 컴파일 결과
 - ✓ `armv5l-linux-gcc -O0 -lrt -o reg_alloc1 reg_alloc1.c`
 - ✓ `armv5l-linux-gcc -O0 -lrt -o reg_alloc2 reg_alloc2.c`
- ✓ 예제 코드에서는 많은 연산을 수행 함으로 일부 어셈블리만 발췌하여 작성하였음.
 - `reg_alloc1` 의 명령어 라인 수
 - 266 line
 - `reg_alloc2` 의 명령어 라인 수
 - 233 line

Register Allocation

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o reg_alloc1 reg_alloc1.c

```
root@parkhk-redhat9:~/embedded_board/optimize/register_allocation/reg_alloc_global
1 00086d0 <reg_Alloc1>:
2
3  ...
4
5  86e8:    e59f03f0    ldr    r0, [pc, #1008]; 8ae0 <reg_Alloc1+0x410>
6  86ec:    e59f13ec    ldr    r1, [pc, #1004]; 8ae0 <reg_Alloc1+0x410>
7  86f0:    e51b2018    ldr    r2, [fp, #-24]
8  86f4:    e51b301c    ldr    r3, [fp, #-28]
9  86f8:    e0822003    add    r2, r2, r3
10 86fc:    e5913000    ldr    r3, [r1]
11 8700:    e0833002    add    r3, r3, r2
12 8704:    e5803000    str    r3, [r0]
13
14  ...
```

함수명

전역 변수를 사용하여
덧셈 연산 수행 (모든
연산은 메모리 접근을
수반함)

1,1 꼭대기

Register Allocation

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o reg_alloc2 reg_alloc2.c



The screenshot shows a disassembler window for the file `reg_alloc2.c`. The assembly code is displayed in a list format with line numbers and addresses. Three specific code blocks are highlighted with colored boxes and annotated with Korean text:

- Blue box (lines 5-9):** `str r0, [fp, -#16]`, `str r1, [fp, -#20]`, `ldr r3, [pc, #876] ; 8a5c <reg_Alloc2+0x38c>`, `ldr r3, [r3]`, `str r3, [fp, -#28]`. Annotation: "전역 변수를 r3에 load 해서 register([fp, -#28] = r5)에 저장" (Load global variable into r3 and store it in register [fp, -#28] = r5).
- Red box (lines 13-18):** `ldr r2, [fp, -#16]`, `ldr r3, [fp, -#20]`, `add r3, r2, r3`, `ldr r2, [fp, -#28]`, `add r2, r2, r3`, `str r2, [fp, -#28]`. Annotation: "register만을 사용하여 덧셈연산을 수행" (Perform addition operation using only registers).
- Blue box (lines 22-24):** `ldr r3, [pc, #108] ; 8a6c <reg_Alloc2+0x39c>`, `ldr r2, [fp, -#44]`, `str r2, [r3]`. Annotation: "전역 변수에 register 값 저장" (Store register value in global variable).

The window title is `root@parkhk-redhat9:~/embedded_board/optimize/register_allocation/reg_alloc_global`. The bottom right corner shows a dropdown menu with "1,5" and "모두" (All).

Register Allocation

■ OO옵션을 이용한 컴파일 결과

- ✓  부분이 제공 연산을 하기 위한 assembly 코드
- ✓  부분은 전역 변수를 지역 변수로 대체하기 위해 데이터를 전달하는 코드
 - 추가적인 작업이지만 전체 수행 코드를 보면 이득임.
 - 전역 변수를 사용한 연산이 적을 때는 손해가 될 수도 있음
- ✓ reg_Alloc1()
 - 전역 변수를 사용함으로써 연산 시 메모리 접근은 수반함
- ✓ reg_Alloc2()
 - 전역변수를 지역 변수로 대체함으로써 전역 변수의 값을 읽어 오거나 연산 후 전역 변수에 값을 저장하는 부가 작업 수행
 - register만을 사용한 연산을 수행함으로써 부가작업으로 인한 오버헤드 보다 더 많은 이득을 얻음.

Register Allocation

- O0옵션을 이용한 컴파일 결과

- ✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 52126000nsec, 39330000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_reg_alloc]$ ./reg_alloc1 ; ./reg_alloc2  
  
reg_Alloc1()  
global_a : 72 global_s : 88 global_m : 8 global_d : 8  
Iteration : 8196 Return Value : -8194 Sum of Random Num. 0  
0sec 52126000nsec  
  
reg_Alloc2()  
global_a : 72 global_s : 88 global_m : 8 global_d : 8  
Iteration : 8196 Return Value : -8194 Sum of Random Num. 0  
0sec 39330000nsec  
[root@ez-x5 dir_reg_alloc]$
```

Register Allocation

■ O3 옵션을 이용한 컴파일 결과

- ✓ `armv5l-linux-gcc -O3 -lrt -o reg_alloc1 reg_alloc1.c`
- ✓ `armv5l-linux-gcc -O3 -lrt -o reg_alloc2 reg_alloc2.c`

- ✓ 예제 코드에서는 많은 연산을 수행 함으로 일부 어셈블리만 발췌하여 작성하였음.
 - `reg_alloc1` 의 명령어 라인 수
 - 152 line
 - `reg_alloc2` 의 명령어 라인 수
 - 89line

Register Allocation

■ O3 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O3 -lrt -o reg_alloc1 reg_alloc1.c

```
root@parkhk-redhat9:~/embedded_board/optimize/register_allocation/reg_alloc_global
1 000862c <reg_Alloc1>:
2
3   ...
4
5   8630:    e59fe264    ldr    lr, [pc, #612]    ; 889c <reg_Alloc1+0x270>
6   8634:    e59f9264    ldr    r9, [pc, #612]   ; 88a0 <reg_Alloc1+0x274>
7   8638:    e59fb264    ldr    fp, [pc, #612]   ; 88a4 <reg_Alloc1+0x278>
8   863c:    e59e4000    ldr    r4, [lr]
9   8640:    e5995000    ldr    r5, [r9]
10  8644:    e59bc000    ldr    ip, [fp]
11  8648:    e0264091    mla    r6, r1, r0, r4
12  864c:    e0803001    add    r3, r0, r1
13  8650:    e0612000    rsb    r2, r1, r0
14  8654:    e0854003    add    r4, r5, r3
15  8658:    e08c5002    add    r5, ip, r2
16  865c:    e58e6000    str    r6, [lr]
17  8660:    e58b5000    str    r5, [fp]
18  8664:    e1a08000    mov    r8, r0
19  8668:    e1a0a001    mov    s1, r1
20  866c:    e5894000    str    r4, [r9]
21
22  ...
23
```

함수명

전역 변수를 사용하여
곱셈 덧셈 뺄셈 연산 수
행 (모든 연산은 메모리
접근을 수반함) 최적화
를 위해 연산 수행의 순
서를 바꿨음.

1,1 모두

Register Allocation

■ O3 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o reg_alloc2 reg_alloc2.c

```
root@parkhk-redhat9:~/embedded_board/optimize/register_allocation/reg_alloc_global
1 000862c <reg_Alloc2>:
2
3   ...
4
5 8630:      e59f8140      ldr    r8, [pc, #320] ; 8778 <reg_Alloc2+0x14c>
6 8634:      e59f5140      ldr    r5, [pc, #320] ; 877c <reg_Alloc2+0x150>
7 8638:      e59f4140      ldr    r4, [pc, #320] ; 8780 <reg_Alloc2+0x154>
8 863c:      e59f2140      ldr    r2, [pc, #320] ; 8784 <reg_Alloc2+0x158>
9 8640:      e5986000      ldr    r6, [r8]
10 8644:      e594c000      ldr    ip, [r4]
11 8648:      e5957000      ldr    r7, [r5]
12 864c:      e0803001      add    r3, r0, r1
13 8650:      e061e000      rsb   lr, r1, r0
14 8654:      e5924000      ldr    r4, [r2]
15 8658:      e0266091      mla   r6, r1, r0, r6
16 865c:      e0875003      add    r5, r7, r3
17 8660:      e1a08000      mov   r8, r0
18 8664:      e1a0a001      mov   sl, r1
19 8668:      e08c700e      add    r7, ip, lr
20
21   ...
22
23 8738:      e59f203c      ldr    r2, [pc, #60] ; 877c <reg_Alloc2+0x150>
24 873c:      e59f303c      ldr    r3, [pc, #60] ; 8780 <reg_Alloc2+0x154>
25 8740:      e0810004      add    r0, r1, r4
26 8744:      e5825000      str   r5, [r2]
27 8748:      e59f2028      ldr    r2, [pc, #40] ; 8778 <reg_Alloc2+0x14c>
28 874c:      e5837000      str   r7, [r3]
29 8750:      e59f302c      ldr    r3, [pc, #44] ; 8784 <reg_Alloc2+0x158>
30 8754:      e5826000      str   r6, [r2]
31
32   ...
```

함수명



전역 변수를 register에 저장

register만을 사용하여 덧셈연산을 수행

전역 변수에 register 값 저장

1,1 꼭대기 Baek

Register Allocation

- O3 옵션을 이용한 컴파일 결과
 - ✓  부분이 제공 연산을 하기 위한 assembly 코드
 - ✓  부분은 전역 변수를 지역 변수로 대체하기 위해 데이터를 전달하는 코드
 - 추가적인 작업이지만 전체 수행 코드를 보면 이득임.
 - 전역 변수를 사용한 연산이 적을 때는 손해가 될 수도 있음
 - ✓ `reg_Alloc1()`
 - 컴파일러에 의해서 코드 최적화
 - 명령어 순서가 바뀜: register에 들어있는 값 재사용
 - ✓ `reg_Alloc2()`
 - 메모리에 있는 전역변수 값을 직접 지역변수에 넣음
 - O0 옵션에서는 다른 register(r3)를 사용하였음.

Register Allocation

- O3옵션을 이용한 컴파일 결과

- ✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 3045000nsec, 2017000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 dir_reg_alloc]$ ./reg_alloc1 ; ./reg_alloc2  
  
reg_Alloc1()  
global_a : 1704 global_s : -57372 global_m : 25015 global_d : 0  
Iteration : 8196 Return Value : -10 Sum of Random Num. 34816  
0sec 3045000nsec  
  
reg_Alloc2()  
global_a : 1704 global_s : -57372 global_m : 25015 global_d : 0  
Iteration : 8196 Return Value : -10 Sum of Random Num. 34816  
0sec 2017000nsec  
[root@ez-x5 dir_reg_alloc]$
```

Register Allocation

- 서브루틴 호출 시 매개변수 처리
 - ✓ 4-레지스터 규칙
 - 4개 이하의 인자를 매개변수로 받는 함수가 5개 이상의 인자를 매개변수로 받는 함수 보다 효율적으로 처리됨
 - 4개의 매개변수는 각각 레지스터 r0~r3로 보내지고 나머지 매개변수는 stack에 쌓이게 됨
 - ✓ 5개 이상의 매개변수를 받아야 하는 경우 구조체를 이용하여 전달하는 것이 효과적임
 - ✓ 컴파일러에 의한 최적화 이유로 구조체를 이용한 매개변수 전달의 최적화가 상쇄될 수 있음

Register Allocation

- Function calls

예제코드

```
/*Function calls*/
main(){
    ...
    for(i=0 ; i<MAX_L; i++){
        func_reg1(init_data[0], init_data[1], init_data[2], init_data[3],
                init_data[4],init_data[5],init_data[6] ,cond);
    }
    ...
}
int func_reg1(int v1, int v2, int v3, int v4, int v5, int v6, int v7, int cond)
{
    v1 &= v2;    v3 |= v1;    v4 &= v3;
    v5 |= v4;    v6 &= v5;    v7 |= v6;

    if(cond > 0)
        return v7;
    return v6;
}
```

Register Allocation

- Function calls

예제코드

```
/*Function calls*/

main(){
    ...
    for(i=0;i<MAX_L;i++){
        func_reg2(init_data[0], init_data[1], init_data[1], ars);
    }
    ...
}

int func_reg2(int v1, int v2, int v3, arg_reg_struct ars)
{
    v1 &= v2;    v3 |= v1;    ars.v1 &= v3;
    ars.v2 |= ars.v1;    ars.v3 &= ars.v2;    ars.v4 |= ars.v3;

    if(ars.cond > 0)
        return ars.v4;
    return ars.v3;
}
```

Register Allocation

■ 예제 코드 비교

✓ 첫 번째 예제 코드

- 정수형 변수 8개를 매개변수로 넘김

✓ 두 번째 예제 코드

- func_reg1()과 수행 내용은 같음.
- 전달되는 매개변수의 개수를 4개 이하로 유지하기 위해 구조체 사용

```
typedef struct reg_struct{
    int v1;
    int v2;
    int v3;
    int v4;
    int cond;
}arg_reg_struct;
```


Register Allocation

■ 00옵션을 이용한 컴파일 결과

- ✓ armv5l-linux-gcc -O0 -lrt -o register_allocation register_allocation.c

```
00008510 <main>:
...
86a8:    da000000    ble     86b0 <main+0x1a0>
86ac:    ea000010    b      86f4 <main+0x1e4>
86b0:    e51b3034    ldr    r3, [fp, #-52]
86b4:    e58d3000    str    r3, [sp]
86b8:    e51b3030    ldr    r3, [fp, #-48]
86bc:    e58d3004    str    r3, [sp, #4]
86c0:    e51b302c    ldr    r3, [fp, #-44]
86c4:    e58d3008    str    r3, [sp, #8]
86c8:    e51b3048    ldr    r3, [fp, #-72]
86cc:    e58d300c    str    r3, [sp, #12]
86d0:    e51b0044    ldr    r0, [fp, #-68]
86d4:    e51b1040    ldr    r1, [fp, #-64]
86d8:    e51b203c    ldr    r2, [fp, #-60]
86dc:    e51b3038    ldr    r3, [fp, #-56]
86e0:    eb000022    bl     8770 <func_reg1>
86e4:    e51b3024    ldr    r3, [fp, #-36]
86e8:    e2833001    add   r3, r3, #1 ; 0x1
86ec:    e50b3024    str   r3, [fp, #-36]
86f0:    eaffffe8    b     8698 <main+0x188>
...

00008770 <func_reg1>:
8770:    e1a0c00d    mov   ip, sp
8774:    e92dd800    stmdb sp!, {fp, ip, lr, pc}
8778:    e24cb004    sub  fp, ip, #4 ; 0x4
877c:    e24dd014    sub  sp, sp, #20 ; 0x14
8780:    e50b0010    str  r0, [fp, #-16]
8784:    e50b1014    str  r1, [fp, #-20]
8788:    e50b2018    str  r2, [fp, #-24]
878c:    e50b301c    str  r3, [fp, #-28]
8790:    e51b2010    ldr  r2, [fp, #-16]
8794:    e51b3014    ldr  r3, [fp, #-20]
...
```

Annotations in the image:

- main함수 (points to 00008510)
- 일부 매개변수를 stack에 저장 (points to the blue highlighted block)
- r0~r3를 이용하여 일부 매개변수를 넘김 (points to the red highlighted block)
- 함수명 (points to 00008770)

Register Allocation

■ O0 옵션을 이용한 컴파일 결과

- ✓ armv5l-linux-gcc -O0 -lrt -o register_allocation register_allocation.c



```
root@parknk-redhat9:~/embedded_board/optimize/register_allocation/function_call_reg
000084dc <main>:
...
8684:    da000000    ble     868c <main+0x1b0>
8688:    ea00000c    b      86c0 <main+0x1e4>
868c:    e1a0c00d    mov    ip, sp
8690:    e24b3044    sub    r3, fp, #68 ; 0x44
8694:    e893000f    ldmia  r3, {r0, r1, r2, r3}
8698:    e88c000f    stmia  ip, {r0, r1, r2, r3}
869c:    e51b3048    ldr    r3, [fp, #-72]
86a0:    e51b0034    ldr    r0, [fp, #-52]
86a4:    e51b1030    ldr    r1, [fp, #-48]
86a8:    e51b2030    ldr    r2, [fp, #-48]
86ac:    eb000022    bl     873c <func_reg2>
86b0:    e51b3024    ldr    r3, [fp, #-36]
86b4:    e2833001    add    r3, r3, #1 ; 0x1
86b8:    e50b3024    str    r3, [fp, #-36]
86bc:    eaffffec    b      8674 <main+0x198>
...
0000873c <func_reg2>:
873c:    e1a0c00d    mov    ip, sp
8740:    e24dd004    sub    sp, sp, #4 ; 0x4
8744:    e92dd800    stmdb  sp!, {fp, ip, lr, pc}
8748:    e24cb008    sub    fp, ip, #8 ; 0x8
874c:    e24dd010    sub    sp, sp, #16 ; 0x10
8750:    e50b0010    str    r0, [fp, #-16]
8754:    e50b1014    str    r1, [fp, #-20]
8758:    e50b2018    str    r2, [fp, #-24]
875c:    e58b3004    str    r3, [fp, #4]
8760:    e51b2010    ldr    r2, [fp, #-16]
8764:    e51b3014    ldr    r3, [fp, #-20]
...
~
~
```

main함수

r0~r3를 이용하여
매개변수를
넘김

함수명

Register Allocation

- OO옵션을 이용한 컴파일 결과
 - ✓  부분은 register를 이용한 인자 전달 과정
 - ✓  부분은 stack을 이용한 인자 전달 과정
 - ✓ 수행 결과(ez-x5)
 - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 8541000nsec, 8434000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 function]$ ./func_reg1 ; ./func_reg2  
  
func_reg1()  
Iteration : 32784  
0sec 8541000nsec  
  
func_reg2()  
Iteration : 32784  
0sec 8434000nsec  
[root@ez-x5 function]$
```

Register Allocation

■ Register Allocation

- ✓ 일반적으로 전역 변수를 사용하여 연산을 수행 하는 것은 부가적인 오버헤드를 유발함.
- ✓ 전역변수의 사용은 메모리 참조 연산을 수반하기 때문에 전역변수를 사용하여 복잡한 연산을 하는 것은 비효율적임
- ✓ 지역변수를 이용하여 연산을 수행하고 그 결과값을 전역변수에 저장하는 것이 효과적임
- ✓ 5개 이상의 매개변수를 받아야 하는 경우 구조체를 이용하여 전달하는 것이 효과적임
- ✓ 컴파일러에 의한 최적화 이유로 구조체를 이용한 매개변수 전달의 최적화가 상쇄될 수 있음

Software Pipelining

■ Software Pipelining

- ✓ 명령어마다 각기 다른 사이클 소요
 - 아키텍처마다 다름
 - ARM9TDMI
 - 덧셈, 뺄셈, 논리 연산과 같은 ALU연산은 한 사이클 소요
 - 곱셈과 나눗셈은 오퍼랜드의 값에 따라 다양한 사이클 소요
 - 분기 명령은 3사이클 소요
 - N개를 저장하거나 로드하는 경우 N 사이클 소요
 - 단 1개 일 경우 2사이클 소요
 - PC를 로드한다면 2사이클 추가 소요
- ✓ 단순화한 ARM9 파이프라인

fetch	decode	ALU	LS1	LS2
--------------	---------------	------------	------------	------------

- Xscale의 경우 7단계 파이프라인
- ARM11의 경우 8단계 파이프라인

LS1, LS2 : load-store 명령관련 작업

Software Pipelining

■ Register Allocation

예제코드

```
/* Register Allocation*/  
  
unsigned long sw_pipeline1(int v1, int v2, int v3, int v4){  
  
    unsigned long i;  
    for(i=0; i<ARRAY_SIZE ; i++){  
        array_v1[i] = v1 + i;  
        array_v11[i] = array_v1[i] & 0xff00;  
        array_v2[i] = v2 + i;  
        array_v12[i] = array_v2[i] & 0xff00;  
        array_v3[i] = v3 + i;  
        array_v13[i] = array_v3[i] & 0xff00;  
        array_v4[i] = v4 + i;  
        array_v14[i] = array_v4[i] & 0xff00;  
    }  
    return i;  
}
```

Software Pipelining

■ Register Allocation

예제코드

```
/* Register Allocation*/  
  
unsigned long sw_pipeline2(int v1, int v2, int v3, int v4){  
  
    unsigned long i;  
    for(i=0; i<ARRAY_SIZE ; i++){  
        array_v1[i] = v1 + i;  
        array_v2[i] = v2 + i;  
        array_v3[i] = v3 + i;  
        array_v4[i] = v4 + i;  
        array_v11[i] = array_v1[i] & 0xff00;  
        array_v12[i] = array_v2[i] & 0xff00;  
        array_v13[i] = array_v3[i] & 0xff00;  
        array_v14[i] = array_v4[i] & 0xff00;  
    }  
    return i;  
}
```

Software Pipelining

■ 예제 코드 비교

✓ sw_pipeline1()

- 배열에 값을 저장한 후 그 값을 바로 다음에 연산에 사용

✓ sw_pipeline2()

- sw_pipeline1()과 같은 결과값을 가짐
- 바로 이전 연산에 종속되지 않도록 작성

Software Pipelining

- O0 옵션을 이용한 컴파일 결과
 - ✓ `armv5l-linux-gcc -O0 -lrt -o sw_pipeline1 sw_pipeline1.c`
 - ✓ `armv5l-linux-gcc -O0 -lrt -o sw_pipeline2 sw_pipeline2.c`
- ✓ 예제 코드에서는 많은 연산을 수행 함으로 일부 어셈블리만 발췌하여 작성하였음.
 - `sw_pipeline1`의 명령어 라인 수
 - 78 line
 - `sw_pipeline2`의 명령어 라인 수
 - 78 line

Software Pipelining

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o sw_pipeline1 sw_pipeline1.c

```
1 0000844c <sw_pipeline1>:
2
3   ...
4
5 8484: e59f00d4 ldr r0, [pc, #212] ; 8560
6 8488: e51b1018 ldr r1, [fp, #-24]
7 848c: e51b2010 ldr r2, [fp, #-16]
8 8490: e51b3018 ldr r3, [fp, #-24]
9 8494: e0823003 add r3, r2, r3
10 8498: e7803101 str r3, [r0, r1, lsl #2]
11 849c: ldr r0, [pc, #192] ; 8564
12 84a0: ldr r1, [fp, #-24]
13 84a4: e59f20b4 ldr r2, [pc, #180] ; 8560
14 84a8: e51b3018 ldr r3, [fp, #-24]
15 84ac: e7923103 ldr r3, [r2, r3, lsl #2]
16 84b0: e7803101 str r3, [r0, r1, lsl #2]
17 84b4: e59f00ac ldr r0, [pc, #172] ; 8568
18 84b8: e51b1018 ldr r1, [fp, #-24]
19 84bc: e51b2014 ldr r2, [fp, #-20]
20 84c0: e51b3018 ldr r3, [fp, #-24]
21 84c4: e0823003 add r3, r2, r3
22 84c8: e7803101 str r3, [r0, r1, lsl #2]
23 84cc: e59f0098 ldr r0, [pc, #152] ; 856c
24 84d0: e51b1018 ldr r1, [fp, #-24]
25 84d4: e59f208c ldr r2, [pc, #140] ; 8568
26 84d8: e51b3018 ldr r3, [fp, #-24]
27 84dc: e7923103 ldr r3, [r2, r3, lsl #2]
28 84e0: e7803101 str r3, [r0, r1, lsl #2]
29   ...
```

함수명

전역변수로 선언된 변수에 인자로 받은 값에 대한 덧셈 결과를 저장

stall 발생

바로 이전에 저장한 변수 값을 이용하여 연산 후 다른 변수에 저장

5,5 모두

Software Pipelining

■ O0 옵션을 이용한 컴파일 결과

✓ armv5l-linux-gcc -O0 -lrt -o sw_pipeline2 sw_pipeline2.c

```
1 000844c <sw_pipeline2>:
2
3   ...
4
5   8484:    e59f00d4    ldr    r0, [pc, #212] ; 8560 <sw_pipeline2+0x114>
6   8488:    e51b1018    ldr    r1, [fp, #-24]
7   848c:    e51b2010    ldr    r2, [fp, #-16]
8   8490:    e51b3018    ldr    r3, [fp, #-24]
9   8494:    e0823003    add   r3, r2, r3
10  8498:    e7803101    str   r3, [r0, r1, lsl #2]
11  849c:    e59f00c0    ldr    r0, [pc, #192] ; 8564 <sw_pipeline2+0x118>
12  84a0:    e51b1018    ldr    r1, [fp, #-24]
13  84a4:    e51b2014    ldr    r2, [fp, #-20]
14  84a8:    e51b3018    ldr    r3, [fp, #-24]
15  84ac:    e0823003    add   r3, r2, r3
16  84b0:    e7803101    str   r3, [r0, r1, lsl #2]
17
18  ...
19
20  84e4:    e59f0084    ldr    r0, [pc, #132] ; 8570 <sw_pipeline2+0x124>
21  84e8:    e51b1018    ldr    r1, [fp, #-24]
22  84ec:    e59f206c    ldr    r2, [pc, #108] ; 8560 <sw_pipeline2+0x114>
23  84f0:    e51b3018    ldr    r3, [fp, #-24]
24  84f4:    e7923103    ldr    r3, [r2, r3, lsl #2]
25  84f8:    e7803101    str   r3, [r0, r1, lsl #2]
26  84fc:    e59f0070    ldr    r0, [pc, #112] ; 8574 <sw_pipeline2+0x128>
27  8500:    e51b1018    ldr    r1, [fp, #-24]
28  8504:    e59f2058    ldr    r2, [pc, #88] ; 8564 <sw_pipeline2+0x118>
29  8508:    e51b3018    ldr    r3, [fp, #-24]
30  850c:    e7923103    ldr    r3, [r2, r3, lsl #2]
31  8510:    e7803101    str   r3, [r0, r1, lsl #2]
32  ...
```

함수명



전역변수로 선언된 변수에 인자로 받은 값에 대한 덧셈 결과를 저장

stall 미발생

이전에 저장한 변수 값을 이용하여 연산 후 다른 변수에 저장

1,1 모두 jeungjae Baek

Software Pipelining

- O0 옵션을 이용한 컴파일 결과
 - ✓  부분이 인자로 받은 값을 전역 변수(배열)에 저장하는 부분
 - ✓  부분은 저장된 전역변수 값을 연산 후 다른 전역 변수(배열)에 저장
 - ✓ `sw_pipeline1()`
 - 이전 연산의 결과 값이 종속적인 코드 생성
 - ✓ `sw_pipeline2()`
 - `sw_pipeline1()`과 같은 결과를 같지만 이전 연산의 결과 값에 비종속적인 코드 생성

Software Pipelining

- OO옵션을 이용한 컴파일 결과

- ✓ 수행 결과(ez-x5)

- 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 605000nsec, 478000nsec의 시간이 걸렸음

```
root@parkhk-redhat9:~  
[root@ez-x5 nfs]$ ./sw_pipeline1_00 ; ./sw_pipeline2_00  
  
sw_pipeline1()  
Iteration : 1024 1024  
0sec 605000nsec  
  
sw_pipeline2()  
Iteration : 1024 1024  
0sec 478000nsec  
[root@ez-x5 nfs]$
```

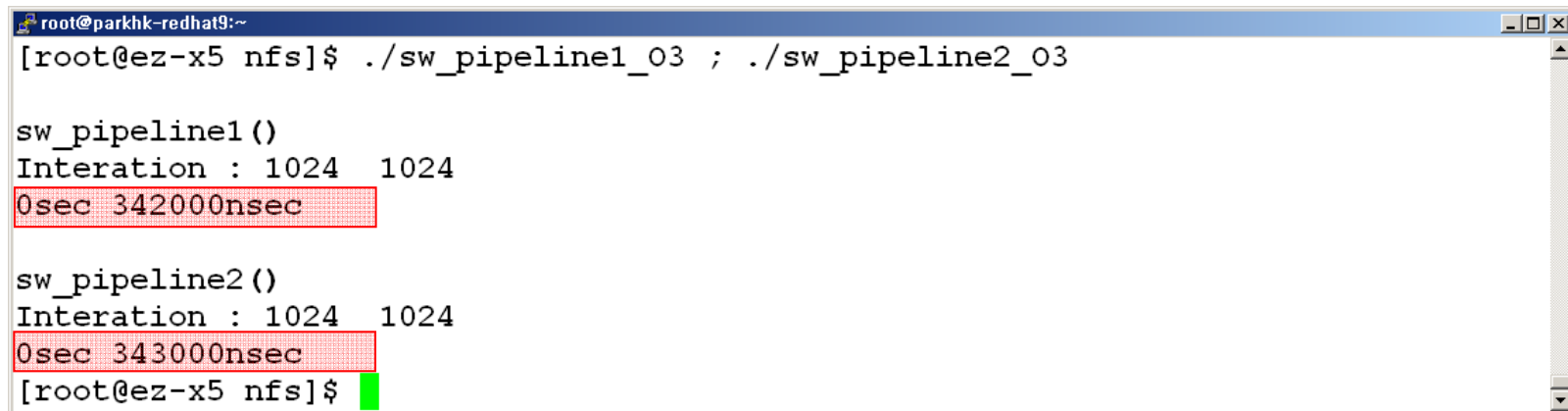
Software Pipelining

■ O3 옵션을 이용한 컴파일 결과

- ✓ `armv5l-linux-gcc -O3 -lrt -o sw_pipeline1 sw_pipeline1.c`
- ✓ `armv5l-linux-gcc -O3 -lrt -o sw_pipeline2 sw_pipeline2.c`

- ✓ 최적화에 의해 같은 코드 생성

- ✓ 수행 결과(ez-x5)
 - 수행결과 같은 결과 값을 갖는 두 개의 함수가 각각 342000nsec, 343000nsec의 시간이 걸렸음(거의 같음)



```
root@parkhk-redhat9:~  
[root@ez-x5 nfs]$ ./sw_pipeline1_03 ; ./sw_pipeline2_03  
  
sw_pipeline1()  
Iteration : 1024 1024  
0sec 342000nsec  
  
sw_pipeline2()  
Iteration : 1024 1024  
0sec 343000nsec  
[root@ez-x5 nfs]$
```

Software Pipelining

■ Software Pipelining

- ✓ 바로 이전 결과값에 종속적인 코드는 data hazard stall을 유발할 수 있음
- ✓ 최적화 옵션에 따라 instruction moving에 의한 최적화로 stall 발생이 방지될 수 있음