Operating System

Assignment #3

담당 교수 : 김태석 교수님
수업 일시 : 월2, 수1
학과 : 컴퓨터 공학과
학번 : 2013722095
이름 : 최 재 은

1. Introduction

우리가 작성하는 프로그램들은 컴파일러를 통해서 assembly code로 변환되고 이는 다시 assembler를 통해서 machine code로 변환 된다. 요즘 대부분의 컴파일러들은 단순한 assembly화가 아닌 코드의 최적화를 같이 수행해 주는데 이러한 과정을 직접 수행해보고 그 결과를 비교해 보는 것을 목적으로 한다.

2. Reference

3. Conclusion

- Analysis

```
segment_id = shmget(1234, PAGE_SIZE, IPC_CREAT | S_IRUSR | S_IWUSR);
shared_memory = (uint8_t*)shmat(segment_id, NULL, 0);

do
{
        shared_memory[i++] = *func;
} while (*func++ != 0xC3);

shmdt(shared_memory);
```

- 위의 D_recompile_test.c 코드를 보면 1234를 key로 갖는 shared memory 공간을 생성하고 그 공간에 func(위에 나오지 않은 Operation 함수)의 body instruction을 저장하는 것을 확인할 수 있다. 이후 현재 프로세스 메모리에 attach했던 공유메모리를 detach한다.

```
default:
        $(CC) -o test2 D_recompile_test.c
        $(CC) -o derecompile D_recompile.c -lrt

dynamic:
        $(CC) -o test2 D_recompile_test.c
        $(CC) -DMYDY -o derecompile D_recompile.c -lrt

dump:
        $(CC) -c D_recompile_test.c
        $ objdump -d D_recompile_test.o > D_recompile_test

clean:
        $(RM) -rf D_recompile_test.o D_recompile_test my_recompile derecompile test2
```

- Makefile을 살펴보면 dynamic의 경우 #ifdef MYDY ~ #endif 의 영역을 포함하여 컴파일이 진행되도록 하였다.
- dump case의 경우 Operation 함수가 있었던 D_recompile_test.c 파일을 compile하여 나온 object file을 다시 recompile하여 파일로 만들어 준다.

```
D_recompile_test.o:     file format elf32-i386


Disassembly of section .text:

00000000 <Operation>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   8b 55 08                mov    0x8(%ebp),%edx
   6:   89 d0                   mov    %edx,%eax
   8:   83 c0 01                add    $0x1,%eax
   b:   83 c0 01                add    $0x1,%eax
   e:   83 c0 01                add    $0x1,%eax
  11:   83 c0 01                add    $0x1,%eax
  14:   83 c0 01                add    $0x1,%eax
  17:   83 c0 01                add    $0x1,%eax
  1a:   83 c0 01                add    $0x1,%eax
  1d:   83 c0 01                add    $0x1,%eax
  20:   83 c0 01                add    $0x1,%eax
  23:   83 c0 01                add    $0x1,%eax
  26:   6b c0 02                imul   $0x2,%eax,%eax
  29:   6b c0 02                imul   $0x2,%eax,%eax
  2c:   6b c0 02                imul   $0x2,%eax,%eax
  2f:   6b c0 02                imul   $0x2,%eax,%eax
  32:   6b c0 02                imul   $0x2,%eax,%eax
  35:   89 c2                   mov    %eax,%edx
  37:   89 55 08                mov    %edx,0x8(%ebp)
  3a:   8b 45 08                mov    0x8(%ebp),%eax
  3d:   5d                      pop    %ebp
  3e:   c3                      ret

0000003f <main>:
  3f:   55                      push   %ebp
  40:   89 e5                   mov    %esp,%ebp
```

- dump된 파일을 열어보면 위와 같은 information이 저장되어있다.
- 일전의 D_recompile_test.c 파일에서는 빨간 박스 영역을 shared memory에 저장한 것이다.
- shared memory에는 빨간 영역 내의 16진수 값이 sequential한 형태로 저장되어 있다.

```
void sharedmem_init()
{
        int shm_id = shmget(1234, PAGE_SIZE, S_IRUSR | S_IWUSR);

        Operation = (char *)shmat(shm_id, NULL, 0);

}
```

- 먼저 컴파일 되지 않은 함수를 가져오기 위해서는 shared memory에 접근해야 한다.
- 1234를 key로 하여 다시 shared memory의 id를 가져오고 현 프로세스에 attach한다.

```
void drecompile_init()
{
        fd = open("my_recompile", O_RDWR|O_CREAT|O_TRUNC, RWMODE);

        compiled_code = mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, fd, 0);
        assert(compiled_code != MAP_FAILED);
}
```

- 가져올 body code를 실행하기 위해서 임의의 파일을 하나 생성한 후 현재 프로세스의 메모리 영역에 mapping해준다.
- 위의 할당할 메모리 영역은 Readable & writable 하다.

```
void* drecompile(uint8_t* func)
{
        int add_count = 0;
        int add_function[3];

        int mul_count = 0;
        int mul_function[3];

        int i = 0;
        int j = 0;
        int k = 0;
        int mul_value = 0x00;

        while(1)
        {
                if(func[i] == 0x00)
                {
                        close(fd);
                        break;
                }

                compiled_code[j] = func[i];

#ifdef MYDY
                if(func[i] == 0x83) // found add inst
                {

                        add_function[0] = func[i];
                        add_function[1] = func[i+1];
```

- 실제 recompile이 진행되는 drecompile 함수에서는 #ifdef 전처리를 사용하여 recompile할 것인지 말 것인지 Makefile에서 선택할 수 있도록 하였다.
- recompile하지 않을 경우에는 그저 shared memory 영역에 있는 값을 그대로 mapping 한 영역으로 가져와 복사하기만 한다. shared memory의 값이 0이면 복사할 값이 없으므로 파일을 닫고 반복문을 종료한다.

```
  mprotect(compiled_code, PAGE_SIZE, PROT_READ | PROT_EXEC); // modify memory permission
  return (void*)compiled_code;
```

- 복사 후에는 mprotect함수를 이용하여 해당 메모리 영역의 권한을 executable하게 바꿔주고 함수를 반환하게 되며, compiled_code가 가리키는 반환된 함수는 main에서 사용이 가능하다.

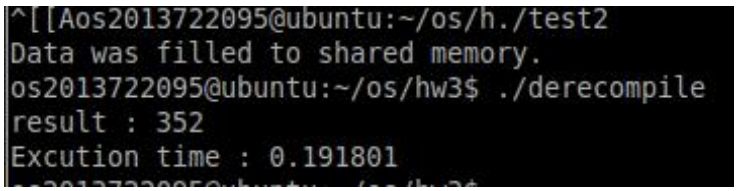
```
if(func[i] == 0x83) // found add inst
{

        add_function[0] = func[i];
        add_function[1] = func[i+1];
        add_function[2] = func[i+2];
        add_count = 1;


        while(1)
        {
                i+=3;
                if(func[i] != add_function[0] || func[i+1] != add_function[1] || func[i+2] != add_function[2])
                        break;

                add_count++;
        }

        compiled_code[j]        = add_function[0];
        compiled_code[j+1]      = add_function[1];
        compiled_code[j+2]      = add_function[2] * add_count;
        j += 3;
        continue;
}
```

- 위의 코드를 살펴보면 처음 ADD(0x83) instruction을 찾으면 그 opcode와 operand들을 array에 저장해두고 count값을 증가시킨다.
- 이후에 반복문을 통해서 현재 저장해둔 instruction과 동일한 instruction이 몇 개나 있는지 개수를 count한다.
- 카운트가 끝나게 되면 그 개수 * operand2의 값을 compiled_code(mapping된 메모리 영역)에 값을 저장하고 그 다음 instruction line으로 이동한다.

```
        else if(func[i] == 0x6b) // found add inst
        {
                mul_function[0] = func[i];
                mul_function[1] = func[i+1];
                mul_function[2] = func[i+2];
                mul_count = 1;


                while(1)
                {
                        i +=3;
                        if(func[i] != mul_function[0] || func[i+1] != mul_function[1] || func[i+2] != mul_function[2])
                                break;

                        mul_count++;
                }

                compiled_code[j] = mul_function[0];
                compiled_code[j+1] = mul_function[1];

                mul_value = mul_function[2];
                for(k = 1; k < mul_count; k++)
                        mul_value *=mul_function[2];

                compiled_code[j+2] = mul_value;

                j += 3;
                continue;
        }
#endif
```

- 위의 ADD에서의 과정과 동일하다. 동일한 imul instruction의 개수를 카운트하고 그 수만큼 operand2의 값을 제곱승하여 mapping된 메모리 영역에 저장한다.
- 위의 그림 왼쪽 하단부를 보게 되면 #endif를 볼 수 있는데 저 지점까지가 최적화 코드의 영역이다.

- 최적화 하지 않은 경우에는 위와 같은 결과를 보이고 최적화시 아래의 그림과 같은 결과를 보였다.



실행 횟수	최적화 X	최적화 O
1	0.3381308	0.191801
2	0.19220727	0.31230993
3	0.836093	0.267649
4	0.535037	0.171308
5	0.885131	0.290173
6	0.182149	0.240411
7	0.158460	0.186004
8	0.178539	0.218273
9	0.587895	0.171808
10	0.196222	0.168200
평균	0.408986407	0.221793693

- 각각 10회의 진행 결과 위와 같은 평균을 보였으며 최적화의 결과가 약 0.18초의 속도 향상을 보였다.

- 고찰

본 과제를 진행하면서 machine code 값 형태로 함수가 저장되어있는 dump file을 보고 딱 들었던 생각은 동일한 명령어에 대해 같은 machine code 값을 가지고 있고 그의 Operand 들 역시 비교 가능한 값으로 표현 되어있다는 것이다. 이를 통해서 특정 instruction (ADD, imul)이 등장하게 되면 그 이후로 Operand까지 완전하게 동일한 함수가 몇 개나 되는지 카운트 한 후에 그에 따라서 Operand에 적절한 곱셈을 해줌으로써 코드의 최적화를 수행 할 수 있겠다는 생각이 들었다. 본 과제를 수행하면서 가장 난해하였던 부분은 단언컨대 memory mapping 부분이었다. 첫째로 mmap가 인자로 file descriptor를 받게 되어있는데, 이 부분에서 공유 메모리 영역을 어떻게 복사할 것인지가 애매했었다. 그래서 갖은 검색을 통해 공유 메모리 영역을 fd형으로 반환받는 shm_open함수까지 사용해보려고 했으나 잘못된 접근임을 깨닫고 수정하였다. 아무튼 빈 파일을 readable/writable하게 생성 후 메모리 영역에 맵핑한 후 그 영역에 공유 메모리에 저장되어 있는 Operation 함수의 body code를 선택적으로 수정하고 복사해온 뒤 해당 메모리 영역을 executable하게 해주어 실행할 수 있게 된다 이 부분에서 메모리 영역을 executable하게 권한 변경해주어야 하는 것을 file의 권한을 실행가능으로 변경하는 것으로 이해하여 꽤 많은 시간을 날려먹었던 기억이 난다.
역시 강의 자료는 꼼꼼하게 읽어야 한다는 생각을 다시금 하게 되었다.