Operating Systems

Assignment #3

Professor	김태석 교수님
Department	컴퓨터공학과
Student ID	2012722028
Name	장 한 별
Date	2018. 12. 04.

Introduction.

이번 과제는 Dynamic Recompilation을 파악해야한다. Dynamic Recompilation은 공유 memory에 존재하는 이미 compile이 완료된 Compiled machine 코드를 가져와서 Optimize하는 방식으로 이루어진다. 특히 이번 과제에서는 불필요하게 반복적으로 연산하는 과정을 최적화 해야하는데 코드가 없고, 실행 파일만 존재할 때 dynamic Recompilation 해서 최적화 하기 전과 후를 비교하는 것을 목표로 한다.

Reference.

- objdump 사용법 참고 http://prkitten.tistory.com/31
- mprotect 함수 참고 https://tip.daum.net/question/116023
- 13학번 최용락 군에게 recompile 된 함수를 실행하는 방법을 터득

문제 해결 과정

Objdump 뜨는 과정

```
default:
	gcc -c D_recompile_test.c
	gcc -o D_recompile_test D_recompile_test.c
	gcc -o D_recompile D_recompile.c -lrt
	objdump -d D_recompile_test.o > test

dynamic:
	gcc -c D_recompile_test.c
	gcc -o D_recompile_test D_recompile_test.c
	gcc -Ddynamic -o D_recompile D_recompile.c -lrt
	objdump -d D_recompile_test.o > test
```

Objdump는 소스 file을 compile한 binary file(object file or execute file)이 있을 때, 이 binary file을 disassemble 할 때 사용한다. Objdump의 사용은 Makefile로 진행했다.

먼저 $ gcc –c D_recompile_test.c 입력 시, Object file이 생성된다.

그 후, $ objdump –d D_recompile_test.o 입력시, D_recompile_test.o file을 disassemble하여 objdump 결과가 출력되는데, 이 부분은 redirection으로 test file에 저장했다. (objump 명령어 중 –d 는disassemble를 의미한다.)

Terminal 에서 'make dynamic'으로 make를 진행할 시 –Ddynamic 옵션을 추가하여 진행한다.

해당옵션이 추가되었을 때 여러 instruction이 중복으로 나올 시, 최적화가 진행되도록 한다.

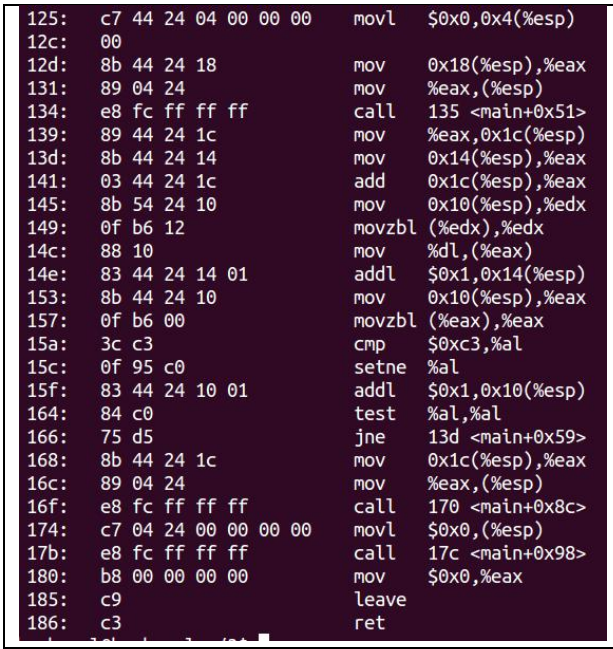
Dump 뜬 파일의 화면

```
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:   6b c0 02                imul   $0x2,%eax,%eax
  11:   6b c0 04                imul   $0x4,%eax,%eax
  14:   83 c0 01                add    $0x1,%eax
  17:   83 c0 01                add    $0x1,%eax
  1a:   83 c0 02                add    $0x2,%eax
  1d:   83 c0 03                add    $0x3,%eax
  20:   83 c0 01                add    $0x1,%eax
  23:   83 c0 02                add    $0x2,%eax
  26:   83 c0 01                add    $0x1,%eax
  29:   83 c0 01                add    $0x1,%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:   83 c0 01                add    $0x1,%eax
  38:   83 c0 01                add    $0x1,%eax
  3b:   83 c0 03                add    $0x3,%eax
  3e:   83 c0 01                add    $0x1,%eax
  41:   83 c0 01                add    $0x1,%eax
  44:   83 c0 01                add    $0x1,%eax
  47:   83 c0 03                add    $0x3,%eax
  4a:   83 c0 01                add    $0x1,%eax
  4d:   83 c0 01                add    $0x1,%eax
  50:   83 c0 02                add    $0x2,%eax
```

```
  53:   83 c0 01                add    $0x1,%eax
  56:   83 c0 01                add    $0x1,%eax
  59:   83 c0 01                add    $0x1,%eax
  5c:   83 c0 01                add    $0x1,%eax
  5f:   6b c0 02                imul   $0x2,%eax,%eax
  62:   6b c0 04                imul   $0x4,%eax,%eax
  65:   83 c0 01                add    $0x1,%eax
  68:   83 c0 01                add    $0x1,%eax
  6b:   83 c0 02                add    $0x2,%eax
  6e:   83 c0 03                add    $0x3,%eax
  71:   83 c0 01                add    $0x1,%eax
  74:   83 c0 02                add    $0x2,%eax
  77:   83 c0 01                add    $0x1,%eax
  7a:   83 c0 01                add    $0x1,%eax
  7d:   6b c0 02                imul   $0x2,%eax,%eax
  80:   6b c0 02                imul   $0x2,%eax,%eax
  83:   6b c0 02                imul   $0x2,%eax,%eax
  86:   83 c0 01                add    $0x1,%eax
  89:   83 c0 01                add    $0x1,%eax
  8c:   83 c0 03                add    $0x3,%eax
  8f:   83 c0 01                add    $0x1,%eax
  92:   83 c0 01                add    $0x1,%eax
  95:   83 c0 01                add    $0x1,%eax
  98:   83 c0 03                add    $0x3,%eax
  9b:   83 c0 01                add    $0x1,%eax
  9e:   83 c0 01                add    $0x1,%eax
  a1:   83 c0 02                add    $0x2,%eax
  a4:   83 c0 01                add    $0x1,%eax
  a7:   83 c0 01                add    $0x1,%eax
  aa:   83 c0 01                add    $0x1,%eax
  ad:   6b c0 02                imul   $0x2,%eax,%eax
  b0:   6b c0 02                imul   $0x2,%eax,%eax
  b3:   6b c0 02                imul   $0x2,%eax,%eax
  b6:   83 c0 01                add    $0x1,%eax
```

```
  b9:   83 c0 01                add    $0x1,%eax
  bc:   83 c0 03                add    $0x3,%eax
  bf:   83 c0 01                add    $0x1,%eax
  c2:   83 c0 01                add    $0x1,%eax
  c5:   83 c0 01                add    $0x1,%eax
  c8:   83 c0 03                add    $0x3,%eax
  cb:   83 c0 01                add    $0x1,%eax
  ce:   83 c0 01                add    $0x1,%eax
  d1:   83 c0 02                add    $0x2,%eax
  d4:   83 c0 01                add    $0x1,%eax
  d7:   83 c0 01                add    $0x1,%eax
  da:   89 c2                   mov    %eax,%edx
  dc:   89 55 08                mov    %edx,0x8(%ebp)
  df:   8b 45 08                mov    0x8(%ebp),%eax
  e2:   5d                      pop    %ebp
  e3:   c3                      ret
```

```
000000e4 <main>:
  e4:   55                      push   %ebp
  e5:   89 e5                   mov    %esp,%ebp
  e7:   83 e4 f0                and    $0xfffffff0,%esp
  ea:   83 ec 20                sub    $0x20,%esp
  ed:   c7 44 24 10 00 00 00    movl   $0x0,0x10(%esp)
  f4:   00
  f5:   c7 44 24 14 00 00 00    movl   $0x0,0x14(%esp)
  fc:   00
  fd:   c7 44 24 08 80 03 00    movl   $0x380,0x8(%esp)
 104:   00
 105:   c7 44 24 04 00 10 00    movl   $0x1000,0x4(%esp)
 10c:   00
 10d:   c7 04 24 d2 04 00 00    movl   $0x4d2,(%esp)
 114:   e8 fc ff ff ff          call   115 <main+0x31>
 119:   89 44 24 18             mov    %eax,0x18(%esp)
 11d:   c7 44 24 08 00 00 00    movl   $0x0,0x8(%esp)
 124:   00
```



```
125:   c7 44 24 04 00 00 00    movl   $0x0,0x4(%esp)
12c:   00
12d:   8b 44 24 18             mov    0x18(%esp),%eax
131:   89 04 24                mov    %eax,(%esp)
134:   e8 fc ff ff ff          call   135 <main+0x51>
139:   89 44 24 1c             mov    %eax,0x1c(%esp)
13d:   8b 44 24 14             mov    0x14(%esp),%eax
141:   03 44 24 1c             add    0x1c(%esp),%eax
145:   8b 54 24 10             mov    0x10(%esp),%edx
149:   0f b6 12                movzbl (%edx),%edx
14c:   88 10                   mov    %dl,(%eax)
14e:   83 44 24 14 01          addl   $0x1,0x14(%esp)
153:   8b 44 24 10             mov    0x10(%esp),%eax
157:   0f b6 00                movzbl (%eax),%eax
15a:   3c c3                   cmp    $0xc3,%al
15c:   0f 95 c0                setne  %al
15f:   83 44 24 10 01          addl   $0x1,0x10(%esp)
164:   84 c0                   test   %al,%al
166:   75 d5                   jne    13d <main+0x59>
168:   8b 44 24 1c             mov    0x1c(%esp),%eax
16c:   89 04 24                mov    %eax,(%esp)
16f:   e8 fc ff ff ff          call   170 <main+0x8c>
174:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
17b:   e8 fc ff ff ff          call   17c <main+0x98>
180:   b8 00 00 00 00          mov    $0x0,%eax
185:   c9                      leave
186:   c3                      ret
```

위 그림은 $ cat test 입력 시, 출력되는 objdump 화면이다. Test file에 redirection을 진행되어 출력된 결과로 위 그림과 같이 objdump가 이루어짐을 확인할 수 있다.

Dump 뜬 파일로부터 어떻게 문제 해결의 실마리를 잡았는지

Objdump 화면과 D_recompile_test.c 에서 assembly 언어로 작성된 부분을 비교하며 힌트를 얻었다. D_recompile_test.c 에서 add 나 imul 명령어가 계속해서 반복됨을 확인할 수 있었는데, 이는 objdump 에서 Add 명령어가 나오면 83, imul 명령어가 나오면 6b가 반복해서 나오는 것을 확인한 결과, Add 명령어는 0x83, imul 명령어는 0x6b로 매칭됨을 확인할 수 있다. 저장 위치는 0xc0이고, 3번째 자리에 위치한 숫자는 얼마만큼의 수를 덧셈이나 곱셈 연산을 수행할지를 확인하는 숫자라는 것 역시 확인할 수 있었다. 이를 이용하여 'make dynamic' 시 dynamic recompilation을 통해 최적화 과정을 진행할 수 있었다.

Shared memory 에서 compiled code 받아오는 부분

```
46  void sharedmem_init()
47  {   // 1
48      int pagesize = 0;
49      pagesize = getpagesize();
50
51      segment_id = shmget(1234, getpagesize(), 0);  // get shared memory
52      Operation = (uint8_t*)shmat(segment_id, NULL, 0);  // attach
53      mprotect(Operation, pagesize, PROT_READ | PROT_WRITE);  // set protection RW
54  }
```

Shared memory에서 compiled code 를 받아오기 위해 먼저 shared memory의 key를 이용하여 shmget() 함수로 공간 요청을 진행했다. 그 후 현재 process가 생성된 shared memory를 사용할 수 있도록 segment id 값으로 attach를 수행 했다. 아직 권한이 주어지지 않았기에, mprotect( )함수로 Read, Write 권한을 부여했다.

Code section

```
void drecompile_init()
{   // 2
    int pagesize = 0;
    pagesize = getpagesize();

    compiled_code = mmap(0, pagesize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);  // memory mapping
}
```

mmap( )를 통해 mapping 시켜주고, 페이지 권한을 위 그림과 같이 설정했다. 그 후 compiled_code는 Read와 Write 권한을 가지고, 다른 process에게 변경 사항을 공유 하는 mapping 방식을 사용했다.

Optimization

```
82    void* drecompile(uint8_t* func)
83    {
84        compiled_code = func;
85
86    #ifdef dynamic          // when make dynamic
87        printf("[dynamic]\n");
```

Code optimization은 dynamic이 설정되어 있을 때 진행된다. 따라서 #ifdef 과 #endif 를 이용해 최optimization이 진행되는 부분과 그렇지 않은 부분을 구분했다..

함수 실행

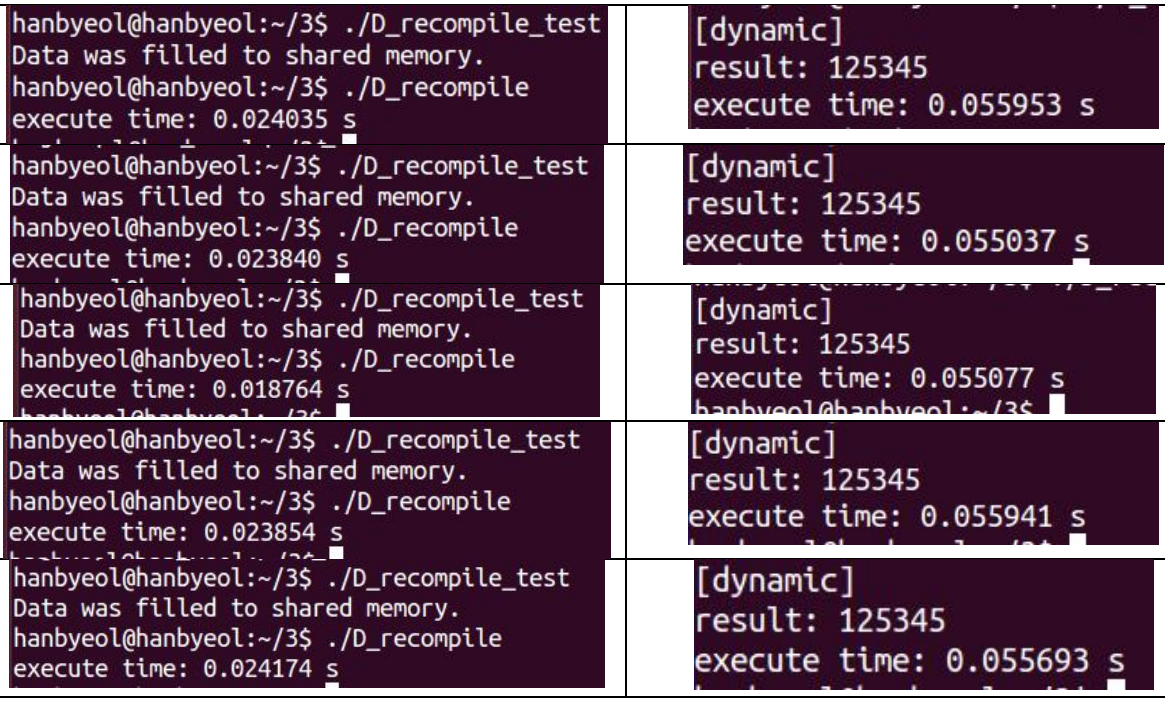
```
func = (int (*)(int a))drecompile(Operation);   // recompile
f_execute = func(0);    // execute function
```

Drecompile( )함수에서 mprotect( )함수로 실행권한을 부여한 후, compiled_code가 반환된다. 이제 해당 함수를 실행할 수 있게 되어 "Conclusion"에서는 optimization 전과 후의 running time을 비교하도록 한다.

Conclusion.

```
hanbyeol@hanbyeol:~/3$ ls
D_recompile.c  D_recompile_test.c  Makefile
hanbyeol@hanbyeol:~/3$ make
gcc -c D_recompile_test.c
gcc -o D_recompile_test D_recompile_test.c
gcc -o D_recompile D_recompile.c -lrt
objdump -d D_recompile_test.o > test
hanbyeol@hanbyeol:~/3$ ls
D_recompile    D_recompile_test    D_recompile_test.o  test
D_recompile.c  D_recompile_test.c  Makefile
```

Make 시 Makefile 에서 default 로 지정해 두었던 명령어로 코드를 object file, execute file을 생성하고, objdump 하여 test file에 redirection 했다.



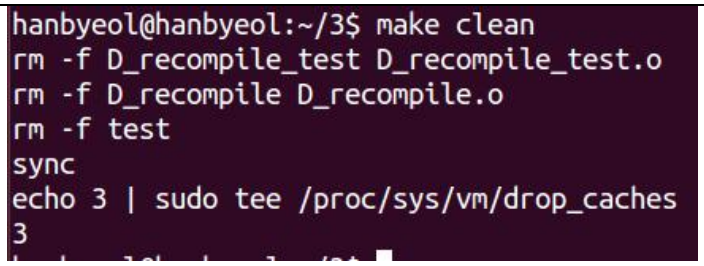
위 그림들 중 왼쪽은 optimization 하지않은 코드를 실행했을 때의 실행 시간을 출력한 화면이고, 오른쪽은 'make dynamic' 명령어를 입력해 dynamic recompilation 으로 optimization 한 후 의 코드를 실행했을 때의 실행 시간을 출력한 화면이다.

Optimization 전: 0.024035s, 0.023840s, 0.018764s, 0.023854s, 0.024174s

평균: 약 0.022933s

Optimization 후: 0.055953s, 0.055037s, 0.055077s, 0.055941s, 0.055693s

평균: 약 0.055540s



물론, 각각의 실행 과정은 다음 과정에 영향을 줄 수 있기 때문에 한번 실행할 때마다 Make clean 명령어를 입력하도록 한다. 그 다음은 'make dynamic'으로 진행한다.

Analysis

Optimization 전의 평균 running time은 약 0.022933s 이고, optimization 후의 평균 running time은 약 0.05540s 이다. Optimization하기 전보다 후의 running time이 오히려 증가한 것으로 보아 optimization이 제대로 이루어지지 않은 것으로 판단된다. 필자는 add 명령어, imul 명령어를 한 줄 씩 읽어나가며 연산하는 방식을 사용했는데, 별로 효율적이지 못한 방법이라 판단된다. 더 빠르게 진행하는 알고리즘을 생각한 후 진행하지 못한 점이 아쉬웠다.