實驗目的

In this project, we provide an isolated and independent module which you can develop your own indirect branch handling optimization routines for the widely used QEMU full-system emulator. You may compare the performance of your optimizations to the original QEMU.
The following sections describe how to build and run the enhanced QEMU and guide you on where to implement your indirect branch handling optimization routines.

實驗器材和所需軟體

- PC x 1
  - Requirement: any x86-based PC.
  - Any linux distribution built for 32-bit x86
  - Qemu, which is provided in this package

Team rule and Due date

1) You may do this assignment in a team of two members, however, you are welcome to do it by yourself.
2) Due date is 2016.4.16.
3) The source code(qemu_virtual_machine.tar.bz2) is in the CEIBA->大綱內容->week 4.

What to Submit?

1) Your source code: wrap up all your codes in a tar file, and upload to CEIBA.

2) A report describes your design and implementation and a discussion section on your experimental results.

Part I. Steps of building & running Qemu

Step 1. Build Qemu

1. Uncompress qemu_virtual_machine.tar.bz2 and get directory ***qemu-0.13.0***

```
$ tar -xf qemu_virtual_machine.tar.bz2
$ ls
qemu-0.13.0    qemu_virtual_machine.tar.bz2
```

2. Create a directory to build qemu

```
$ mkdir build.qemu
$ ls
build.qemu    qemu-0.13.0    qemu_virtual_machine.tar.bz2
```

3. Enter the build directory and build the qemu

```
$ cd build.qemu/
$ ../qemu-0.13.0/configure --target-list=i386-linux-user
(… some output …)
$ make
(… some output …)
```

4. The qemu executable is located at ***i386-linux-user/qemu-i386*** after make

```
$ ls i386-linux-user/qemu-i386
i386-linux-user/qemu-i386
```

Step 2. Run Qemu

1. Run 32-bit executable in the command "**qemu-i386 ${PROGRAM NAME}**," above is an example executing ls

Ex. (Assuming the /bin/ls is a 32-bit, 80386 executable)

$ ***i386-linux-user/qemu-i386 /bin/ls***
(.. content of current directory …)

2. You can use command ***time*** to measure the execution time

Ex. (Assuming the /bin/ls is a 32-bit, 80386 executable)

$ ***time*** *i386-linux-user/qemu-i386 /bin/ls*
(.. content of current directory …)

real 0m0.029s
user 0m0.016s
sys 0m0.008s

3. You can also use ***perf*** to exploit more information of the execution

Ex. (Assuming the /bin/ls is a 32-bit, 80386 executable)

$ ***perf stat*** *i386-linux-user/qemu-i386 /bin/ls*
(.. content of current directory …)

Performance counter stats for 'i386-linux-user/qemu-i386 /bin/ls':

(... measurement of the execution ...)

0.050433800 seconds time elapsed

Part II. Optimizations of indirect branch handling

Step 1. Optimization Suggestions

In the provided module, we have already put some function hooks to help implementing two indirect branch handling mechanisms: a shadow stack and an indirect branch target cache. These function hooks help you focusing on designing and implementing the algorithm of the indirect branch handing mechanisms. The function definition can be found in **#<qemu_dir>/optimization.c**, and the prototype of the functions are already provided. You can leverage these prototypes to implement the indirect branch handling routines. Below are some suggestions about how to implement these function hooks.

Step 2. Shadow Stack

A shadow stack is used to accelerate searching branch targets when current instruction is a return instruction. The address of the translation block corresponding to next instruction of the function call is pushed on the shadow stack while the function call is executed. When the callee returns, the top of the shadow stack is popped, and the popped address is the translation block of the return address.

Data structure of shadow stack:

(In **cpu-defs.h**)

```
uint64_t *shack;
uint64_t *shack_top;
uint64_t *shack_end;
void *shadow_hash_list;
int shadow_ret_count;
unsigned long *shadow_ret_addr;
```

The variables listed above are useful to implement the shadow stack. They are data fields in the data structure CPUState. CPUState is used to record the current architecture

state of the emulated platform. The field **shack** points to the memory area that contains the stack entries, and the **shack_top** and **shack_end** are used to implement the push and pop operations.

The field **shadow_ret_addr** is used to store a list of the host return addresses. Due to the translation block of the target return address may not be translated when it is pushed onto the runtime stack, maintaining the host return addresses can help you implement the shadow stack easier. The field **shadow_hash_list** is a hash table with chaining mechanism which can help finding the unknown host return address described above

```
(In optimization.h)
struct shadow_pair
{
        struct list_head l;
        target_ulong guest_eip;
        unsigned long *shadow_slot;
};
```

The structure **shadow_pair** is the hash entry of the hash table **shadow_hash_list**. It is implemented with intrusive list (**struct list_head l**) and the field **l** is used to perform hash chaining when collision happened. The field **guest_eip** represented as the guest return address and the field **shadow_slot** will later be filled in the host return address.

This data structure is used when the translation block of the guest return address is yet to be created. This situation will be further described when introducing the push action.

Function hooks:

1. Initial shadow stack

```
void shack_init (CPUState *env);
```

In the function **shack_init**, you may allocate the shadow stack and initial self-defined structure to implement shadow stack. For example, you need to allocate a memory space which will be later used to store the stack entries, and the allocated memory should be kept in the shadow stack related fields in the **CPUState** structure. Both the **shack_top** and **shack_end** should be set to the beginning and the end of the allocated memory.

```
/* Create shadow stack. */
env->shack = (uint64_t *)malloc(SHACK_SIZE * sizeof(uint64_t));
```

Fig : Allocate shadow stack

The field **shadow_hash_list** and **shadow_ret_addr** are also required to allocate a memory space for further use. Noticeably that the number of entries in the allocated area pointed to by **shadow_ret_addr** should be equal to the number of entries in shadow stack since the related host return address is stored in the allocated area that **shadow_ret_addr** points to.

2. Push shadow stack

```
void push_shack (CPUState *env, TCGv_ptr cpu_env, target_ulong next_eip);
```

In the function **push_shack** you need to implement the push operation of the shadow stack. It includes pushing two addresses: the guest return address and the host return address. The guest return address will be used to confirm the same branch target in the pop action, and the parameter **next_eip** contains the guest return address. The host return address is the beginning of the translated code sequence, it can be get by the host return address. For more detail you can refer to the function **tb_find_slow** in **#<qemu_dir>/cpu-exec.c**, and it shows how to get the corresponding translation block by the guest return address. The host return address can be access through the translation block.

The push operation needs to be generated as the target machine code since it pushes whenever a function call is invoked. To generate machine code in the qemu, you need to get familiar with the **tiny code generator (tcg)** and implement these functions by the help of some **tcg** functions. The detail of how tcg functions work can be refer to the source file **#<qemu_dir>/tcg/tcg-op.h** and **#<qemu_dir>/tcg/tcg.c.** You can find more information here.

Here are some tcg operations can help you finish the **push_stack**:

Create temporary variable:

```
TCGv tcg_temp_new()
TCGv tcg_temp_new_ptr()
```

Release temporary variables
void tcg_temp_free(TCGv)
void tcg_temp_free_ptr(TCGv)

Load and store memory address
void tcg_gen_ld_tl(TCGv, TCGv_ptr, tcg_target_long)
void tcg_gen_ld_ptr(TCGv, TCGv_ptr, tcg_target_long)
void tcg_gen_st_tl(TCGv, TCGv_ptr, tcg_target_long)
void tcg_gen_st_ptr(TCGv, TCGv_ptr, tcg_target_long)

Control structures
int gen_new_label()
void tcg_gen_brcond_tl(TCGCond, TCGv, TCGv, int)
void gen_set_label(int)

Arithmetic
void tcg_gen_addi_ptr(TCGv, TCGv, int32_t)

(Please find the definitions in the **tcg/tcg-op.h** & **tcg/tcg.c**)

Here is an example that pushes next pc on the stack top using tcg API:

```
TCGv_ptr temp_shack_top

tcg_gen_ld_ptr(temp_shack_top, cpu_env,          // load next pc
                    offsetof(CPUState, shack_top));
tcg_gen_st_tl(tcg_const_tl(next_eip),            // store to stack top
                  temp_shack_top, 0);
```

In the example listed above, the macro **offsetof** can help to access the specific field in a data structure. Due to some of the shadow stack related variables are data fields in the CPUState. You can use the macro to operate on the fields in the CPUState to manipulate the shadow stack.

Flush Shadow Stack

Besides of implementing the push operation, you also need to check if the stack is full while pushing a new element on the stack. The stack can be flush in various ways, and the only requirement is the pop stack should work correctly on a flushed stack. To check

if the stack is full you need to generate conditional check code by tcg, and here is a tcg example that creates a new label and conditional branches to the label:

```
int label = gen_new_label();                    // create label
tcg_gen_brcond_ptr(TCG_COND_NE,                 // branch to label
                   lhs, rhs, label);

…other tcg operations…

gen_set_label(label);                           // set label here
```

Push guest_eip doesn't have corresponding translation block

One other thing you should be aware of is the guest eip may not have corresponding translation block since the address has not been executed. This results in unknown host return address. Since the host return address may be unknown while pushing, you need first record the pushed guest return address. When the translation block of the guest return address is translated, the host return address should be update to the shadow stack.

There are two things you needs to do, the first is backup the unresolved host return address. You can use the data field **shadow_hash_list** in CPUState and the data structure **shadow_pair** to build up a hash table, which makes it easier to find the corresponding slot of the host return address by the guest return address.

The second is updating the corresponding slot when the translation block is created. The function **shack_set_shadow** will be called when a new translation block is created. It is called from function **cpu_gen_code** in **#<qemu_dir>/translate-all.c**. You should get the corresponding slot by the guest return address (guest_eip) and update it by the host return address (host_eip.) The data field **shadow_hash_list** in CPUState and the data structure **shadow_pair** can help you finish the job.

You may need such conditional check and assignment to finish setting shadow stack:

```
(in shack_set_shadow)
…other works…
struct shadow_pair *sp = #<some source>

…other works…
```

```
if (sp->guest_eip == guest_eip)
{
        *sp->shadow_slot = (uintptr_t)host_eip;
        …other works…
}

…other works…
```

3. Pop shadow stack

```
void pop_shack (TCGv_ptr cpu_env, TCGv next_eip);
```

In the function **pop_shack** you need to implement the pop operations of the shadow stack. It also requires tcg APIs to generate the machine code. You can refer to the list of tcg API described above to finish the job.

At the pop operation side, you should check if the next guest address is equal to the guest return address of the top entry on the shadow stack. The next guest address is the parameter **next_eip**. Sometimes the guest return address may not equal to the next guest address due to flush of stack, and the pop_shack can only return to the host return address when the guest return address is equal to the next guest address.

To insert machine code to jump to the target address in the **pop_shack**, you can use the following code template:

```
*gen_opc_ptr++ = INDEX_op_jmp;
*gen_opparam_ptr++ = #<Target Address>;
```

The target address in the provided template can be replaced by host return address in the pop_shack function.

Step 3. Indirect Branch Target Cache

Indirect branch target cache works similar to hardware cache, but it stores the address of the related code fragment stored in the code cache. Each time a cache lookup

succeeds, the indirect branch can directly branch to the target address but not the emulation engine.

Data structure of indirect branch target cache:

```
struct jmp_pair
{
    target_ulong guest_eip;
    TranslationBlock *tb;
};
```

Structure **jmp_pair** represents the cache entry of the indirect branch target cache, field **guest_ip** represents the key and **tb** is the translation block to be executed.

```
struct ibtc_table
{
    struct jmp_pair htable[IBTC_CACHE_SIZE];
};
```

Structure **ibtc_table** represents the cache.

```
#define IBTC_CACHE_MASK      (IBTC_CACHE_SIZE - 1)
uint8_t *optimization_ret_addr;
```

Constant value **IBTC_CACHE_MASK** can be used to calculate the index with the **guest_eip**. The pointer **opimizetion_ret_addr** can help you exit cache lookup when the lookup failed.

Function hooks:

```
void ibtc_init (void);
```

In function ibtc_init, you need to allocate a space for an ibtc_table. To ensure the correct of later execution, please set the allocated area to zero.

```
void *helper_lookup_ibtc (target_ulong guest_eip);
```

In function **helper_lookup_ibtc**, you can get the related cache entry by the argument guest_eip. If the lookup success, you should return the context pointer of the related

translation block. Otherwise, you can return the optimization_ret_addr to return to the emulation engine.

```
void update_ibtc_entry (TranslationBlock *tb);
```

In function **update_ibtc_entry**, you should update the translation block and the guest address to the related cache entry. Noticeably that the function **helper_lookup_ibtc** is executed before the function **update_ibtc_entry**, so the information such as guest_ip can be preserved in function **helper_lookup_ibtc** and used in function **update_ibtc_entry** later.

Part III. Benchmark

Step 1. Recommended Benchmarks

Before measuring the *correctness* and *effectiveness* of your design and implementation, you should contrive some small test cases. Do remember that your design is for "Indirect Branch" handling, so that your test cases must contain such "Indirect Branches". After the sanity check of your implementation, you may start to use a set of benchmarks to evaluate whether your implementation is for real! Here are some benchmarks we recommend you to consider measuring the performance:

MiBench:
http://www.eecs.umich.edu/mibench/source.html

CoreMark:
http://www.coremark.org/download/index.php?pg=download