Virtual Machine - Assignment 1

R04922058 鄭以琳

April 14, 2016

1 Shadow Stack

1.1 Data Structure

```
//cpu-defs.h
void *shack;
void *shack_top;
void *shack_end;
struct hash_list *shadow_hash_list;
```

Variables `shack`, `shacks_top` and `shack_end` are the same as which defined in homework guidelines. However, I modified `hash_list`. It now contains several buckets, each pointing to the head of a `shadow_pair` chain.

```
//optimization.h
struct shadow_pair{
    target_ulong guest_eip;
    unsigned long *host_eip;
    struct shadow_pair *nxt;
};

struct hash_list{
    struct shadow_pair *head[SHACK_BUCKET];
};
```

In a `shadow_pair`, `guest_eip` represents the guest return address, and `host_eip` represents the host return address. `nxt` is a pointer points to the next `shadow_pair` when hash collision occurs.

`hash_list` contains 2^{16} entries (defined as `SHACK_BUCKET` in `optimization.c`). `head` holds the pointers of the first elements of the entries.

When function `push_shack()` is called, the address of the corresponding `shadow_pair` will be pushed onto the stack. A `shadow_pair` will be updated if `shack_set_shadow()` is called. The function will use `shadow_hash_list` to get the corresponding `shadow_pair` and update the host return address.

1.2 Functions

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

In this function, shadow stack and hash list are initialized.

```
shadow_pair* find_hash_pair(CPUState *env, target_ulong next_eip)
```

This is a function that helps finding the conrresponding `shadow_pair` with a given guest return address. If no corresponding `shadow_pair` is found, it will create a new one and return the address.

There are 2^{16} entries in the hash list. Thus the 16 least significant bits of `next_eip` are used as hash key.

When a hash collision occurs, the new `shadow_pair` is added to the head of the chain of the corresponding entry, and `hash_list->head` is modified to point to the new `shadow_pair`.

```
void shack_set_shadow(CPUState *env, target_ulong guest_eip, unsigned long
   *host_eip)
```

When a new translation block is created, this function will be called. It uses `find_hash_pair()` to retrieve the `shadow_pair` corresponding to `guest_eip`, and update `host_eip` in the `shadow_pair`.

```
void helper_shack_flush(CPUState *env)
```

This function flushes the entire shadow stack by setting `shack_top` to `shack`.

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

The push operation contains several steps.

1. Get the corresponding `shadow_pair`.
   Call `find_hash_pair()` to retrieve the `shadow_pair` that contains `next_eip`. Since the address of the corresponding `shadow_pair` will not change, this step does not need to be written in TCG. The address retrieved at translation time will be used to generate TCG codes in the following steps.

2. Check if need to flush the stack.
   If `shack_top` equals to `shack_end`, the stack is full and needs to be flushed. Flushing simply sets `shack_top` to `shack`.

3. Push the address of `shadow_piar` onto the stack.
   The address of `shadow_pair` retrieved in the first step will be pushed onto the stack.

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

The pop operation contains several steps.

1. Check if the stack is empty.
   If `shack_top` equals to `shack`, the stack is empty and no other operations need to be performed.

2. Check if the guest address of the top entry matches `next_eip`.
   If the `shadow_pair` on top of the shack contains the `guest_eip` same as `next_eip`, go to the next step; otherwise no other operations need to be performed.

3. Check if the host address of the top entry is valid.
   If the `shadow_pair` contains a valid host address (not `NULL`), go to the next step; otherwise no other operations need to be performed.

4. Update return address.
   Update the return address by modifying `gen_opc_ptr` as stated in homework guildelines.

2 Indirect Branch Target Cache

2.1 Data Structure

```
//optimization.h
struct jmp_pair{
    target_ulong guest_eip;
    TranslationBlock *tb;
};

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

IBTC is implemented using a direct map. Unlike the hash list used in shadow shack, each entry can only hold a single value instead of a chain. When a hash collision occurs, only the new value will be saved.

```
//optimization.c
__thread int update_ibtc;
struct ibtc_table *ibtc;
target_ulong saved_eip;
```

`update_ibtc` is a flag indicating whether `update_ibtc_entry()` should be called in `cpu-exec.c`. Since `helper_lookup_ibtc()` is always executed before `update_ibtc_entry()`, I use `saved_eip` to preserve the guest address recieved in function `helper_lookup_ibtc()`.

2.2 Functions

```
void ibtc_init(CPUState *env)
```

In this function, `ibtc_table` is initialized.

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

When there is an indirect jump instruction, this function helps to look up IBTC. If cache hits, return the saved host address. Otherwise, set flag `update_ibtc` to 1, set `saved_eip` = `guest_eip` and return `optimization_ret_addr`.

Since `ibtc_table` holds 2^{16} entries, the 16 least significant bits of `guest_eip` are used as hash key.

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

This function will be called if flag `update_ibtc` is set. It will insert an entry holding `saved_eip` and the corresponding `translationBlock` into `ibtc_table`. If a collision occurs, only the newly created entry will be preserved.

3 Experiment

I ran two benchmarks MiBench and CoreMark to measure the correctness and effectiveness of my implementation.

3.1 Environment

- Virtualbox Ubuntu 32-bit
- 1 core, Intel(R) Xeon(R) CPU E3-1230 V2 @ 3.30GHz
- 2G memory

3.2 MiBench

MiBench provides several benchmarks. I chose `automotive` to run the experiments. There are 4 programs in `automotive` benchmark. I ran each program five times and calculated the average runtime and speedup.

The commands used are:

```
> qemu-i386 $BM_PATH/basicmath_large 1>/dev/null

> qemu-i386 $BM_PATH/bitcnts 10000000 1>/dev/null

> qemu-i386 $BM_PATH/qsort_large $BM_PATH/input_large.dat 1>/dev/null

> qemu-i386 $BM_PATH/susan $BM_PATH/input_large.pgm /dev/null -s
  1>/dev/null
```

The results are shown below.

Table 1: MiBench - automotive

	Without OPT	**With IBTC**		**With shadow stack**		**With Both OPT**	
	Runtime(s)	Runtime(s)	Speedup	Runtime(s)	Speedup	**Runtime(s)**	**Speedup**
basicmath	4.116	3.9948	1.03	3.3972	1.21	**3.3862**	**1.22**
bitcount	6.4718	4.3346	1.49	4.536	1.43	**2.8492**	**2.27**
qsort	0.7048	0.7172	0.98	0.6648	1.06	**0.6366**	**1.11**
susan	0..3236	0.4104	0.78	0.3912	0.83	**0.3776**	**0.86**

3.3 CoreMark

I downloaded `CoreMark_v1.0` to do the experiments. It provides a single test program. I ran it with two sets of different parameters suggested by its Readme file.

```
> qemu-i386 $BM_PATH/coremark.exe 0 0 0x66 0 7 1 2000

> qemu-i386 $BM_PATH/coremark.exe 0x3415 0x3415 0x66 0 7 1 2000
```

Each set of parameters is ran three times and the average runtime with their speedup are listed below.

Table 2: CoreMark

	Without OPT	**With IBTC**		**With shadow stack**		**With Both OPT**	
	Runtime(s)	Runtime(s)	Speedup	Runtime(s)	Speedup	**Runtime(s)**	**Speedup**
Test case 1	15.13	12.89	1.17	14.40	1.05	**12.18**	**1.24**
Test case 2	15.3	13.06	1.17	14.33	1.07	**12.17**	**1.26**

3.4 Discussion

In most of the cases, implementing IBTC and shadow stack leads to better performance. However, since both optimizations are designed for indirect branch handling, the effectiveness is not so obvious if the test case only contains a few such branches.