Shadow Stack and Indirect Branch Translation Cache Laboratory

CSIE b00902107 Shu-Hung, You

(Note: the program only runs on 32-bit linux)

1 Implementation

1.1 Shadow Stack

The data structures used in shadow stack includes

- **shadow stack**, a stack that consists of $(guest\_eip, slot\_pointer)$ pairs.
- **hash table**, h, a mapping taking *guest_eip* to *slot_pointer*. i.e. $h(guest\_eip) = slot\_pointer$
- **slot**s, lots of memory cells that contain the host return addresses

All *slot_pointer* points to some memory cell storing the host return address. In our implementation, the `pop_shack` function will obtain the host return address by reading the slot pointed by *slot_pointer* on the shadow stack. Upon a new translation block is created, the *shack_set_shadow* will update the slot pointed by *slot_pointer* in the hash table.

To be precise,

- `shack_set_shadow` Lookup for *slot_pointer* in the hash table by *guest_eip*. If an item is found, say $h(guest\_eip) = ptr$, then set $*ptr \leftarrow host\_eip$.
- `push_shack`
  1. Code generation phase: If the shadow stack is full, flush it by resetting the stack pointer. Lookup for *slot_pointer* in the hash table. If not found, allocate a new slot.
  2. Runtime: Push the item $(guest\_eip, slot\_pointer)$ to the shadow stack (where the $h(guest\_eip) = slot\_pointer$. This is pre-calculated in code generation phase.)
- `pop_shack` If the *guest_eip* in shadow stack matches and if $*slot\_pointer$ is non-null, pop the stack and jump to $*slot\_pointer$. Otherwise do nothing.

1.2 IBTC

We simply creates a direct mapped cache taking *guest_eip* to *host_eip*. The implementation is the same as specified in the homework. When `helper_lookup_ibtc` lookup failed, we set the global variable `update_ibtc` to 1 to update the cache.

1.3 Call Cache

This is almost the same as the IBTC cache except that

- It is set in `push_shack`
- It is also updated in `shack_set_shadow`

In `pop_shack` we simply lookup the cache.

1.4 Simple Shadow Stack

The is similar to the full shadow stack, yet the hash table and the slot are removed. When pushing to the shadow stack, we try to find the address as in the function `tb_find_slow`. If found, we directly push the *guest_eip* and *host_eip* to the shadow stack. If not found, we do nothing.

2 Benchmarks

2.1 Specialized Test

We have test our program using very simple code to make sure that our optimization is working.

1. shadow stack test and flushing test

```
global _start
[BITS 32]
_start:
  mov ecx, 10000000
_loop:
  call _func
  loop _loop
finish:
  xor eax,eax
  xor ebx,ebx
  inc eax
  int 0x80
_func:
  ret
```

```
global _start
[BITS 32]
_start:
  xor eax,eax
  call _func
  mov ecx, 200
_loop:
  mov eax, 100000
  call _func
  loop _loop
finish:
  xor eax,eax
  xor ebx,ebx
  inc eax
  int 0x80
_func:
  or eax,eax
  jz .bye
  dec eax
  call _func
.bye:
  ret
```

2. IBTC test

```
global _start
[BITS 32]
_start:
  mov ecx, 20000000
  call _func
_func:
  pop eax
  sub esp, 4
  dec ecx
  jz  .exit
  jmp eax
.exit:
  xor eax,eax
  xor ebx,ebx
  inc eax
  int 0x80
```

2.2 Mibench

We have tested the optimizations using the Mibench benchmark suite against QEMU with

- *qemu-noopt*, no optimization
- *qemu-defhw*, the old shadow stack specification in the homework
- *qemu-shack*, full shadow stack and IBTC
- *qemu-nopush-cache*, simple shadow stack, call cache and IBTC
- *qemu-nopush*, simple shadow stack and IBTC
- *qemu-cache*, call cache and IBTC

The benchmark is performed by running each program 5 times and removing the extremum running time.

	qemu-noopt	qemu-defhw	qemu-shack	qemu-nopush-cache	qemu-nopush	qemu-cache
dijkstra	0.150	0.157	0.150	0.150	0.153	0.150
patricia	1.147	1.167	0.977	0.983	1.120	0.967
stringsearch	0.010	0.010	0.010	0.010	0.010	0.010
rawcaudio large.pcm	0.623	0.627	0.633	0.630	0.613	0.630
rawdaudio large.adpcm	0.907	0.730	0.737	0.720	0.727	0.727
crc	2.377	1.763	1.657	1.690	1.683	1.817
fft	0.610	0.617	0.510	0.513	0.647	0.500
fft -i	0.420	0.423	0.350	0.350	0.410	0.350
toast	1.090	1.100	1.050	1.040	1.090	1.043
untoast	0.417	0.420	0.360	0.370	0.400	0.363
basicmath_large	4.520	4.453	3.850	3.807	4.377	3.810
bitcnts	0.753	0.387	0.380	0.390	0.380	0.393
qsort_large	0.673	0.637	0.560	0.550	0.610	0.580
susan -s	0.433	0.440	0.440	0.430	0.433	0.430
susan -e	0.080	0.080	0.080	0.080	0.080	0.080
susan -c	0.020	0.030	0.030	0.030	0.030	0.033
rijndael e	0.350	0.340	0.320	0.313	0.333	0.310
rijndael d	0.340	0.327	0.310	0.310	0.327	0.307
sha	0.143	0.140	0.140	0.140	0.140	0.140
lout	0.660	0.653	0.590	0.610	0.637	0.580
cjpeg	0.090	0.090	0.090	0.090	0.090	0.090
djpeg	0.040	0.040	0.040	0.040	0.040	0.040
tiff2bw	0.273	0.263	0.257	0.250	0.253	0.253
madplay	0.597	0.600	0.570	0.580	0.590	0.577
tiff2rgba	0.663	0.660	0.690	0.660	0.590	0.710
tiffmedian	0.510	0.513	0.510	0.510	0.510	0.503
lame	8.927	8.950	8.833	8.783	8.857	8.847
tiffdither	0.837	0.833	0.790	0.800	0.820	0.790

3 Discussion

As can be seen in the result, the call cache optimization runs roughly as fast as the full shadow stack approach in most test cases. When combined with simple shadow stack, some programs ran faster while some slower. The `fib` program, however, runs much slower when the shadow stack optimization is no available.

The simple shadow stack optimization alone also improves the performance. It is clearly slower compared to the full shadow stack implementation, but the full implementation does not speed up the program very much. This might due to the overhead of indirect memory load.

Even though the call cache optimization and simple shadow stack alone speed up the program, their combination is somewhat mystery and has no obvious improvement, and the full shadow stack approach almost works the best among the optimizations.

Surprisingly, the old specification in the homework (which requires us to modify the shadow stack directly in `shack_set_shadow`) does improve performance (though not much) in some test cases. This might be caused by function call inside loops so the in the latter runs the code would have been translated.