CS510 Final Project

Part 1. L2 Data TLB

A. Explanation of implementation

```
# 2nd-Level TLB Option
parser.add_option("--l2dtb", action="store_true")
parser.add_option("--l2dtb_size", type='int', default=-1)
parser.add_option("--l2dtb_assoc", type='int', default=-1)
```

[configs/common/Options.py]

Three parameters determine the size and associativity of L2 TLB, and also whether to use it or not. If you set **--l2dtb** option to use L2 TLB, **--l2dtb_size** and **--l2dtb_assoc** should also be mentioned together, otherwise L2 TLB will not be created. Rather than setting the parameter directly, I recommend to use existing shell file: **run_part1.sh**.

```
# 2nd-Level TLB
for i in range(options.num_cpus):
    if options.l2dtb:
        system.cpu[i].dtb.l2_size = options.l2dtb_size
        system.cpu[i].dtb.l2_assoc = options.l2dtb_assoc
```

[configs/common/CacheConfigs.py]

CacheConfigs.py defines cache-related configurations for each CPU. I add some code here that initializes the size and associativity of L2 data TLB if needed. Here, **dtb.l2_size** and **dtb.l2_assoc** come from the variables of parser in previous step.

```
from m5.objects.X86TLB import X86TLB as ArchDTB, X86TLB as ArchITB
    dtb = Param.BaseTLB(ArchDTB(), "Data TLB")
    itb = Param.BaseTLB(ArchITB(), "Instruction TLB")
```

[src/cpu/BaseCPU.py]

```
class X86TLB(BaseTLB):
    type = 'X86TLB'
    cxx_class = 'X86ISA::TLB'
    cxx_header = 'arch/x86/tlb.hh'
    size = Param.Unsigned(64, "TLB size")
    l2_size = Param.Int(-1, "2nd-level TLB size")
    l2_assoc = Param.Int(-1, "2nd-level TLB associativity")
    system = Param.System(Parent.any, "system object")
    walker = Param.X86PagetableWalker(\
            X86PagetableWalker(), "page table walker")
```

[src/arch/x86/X86TLB.py]

In **BaseCPU.py**, it is able to find that instruction and data TLB belongs to **X86TLB** class. Hence, I add two variables: **l2_size** and **l2_assoc** in **X86TLB** class located in **X86TLB.py**.

```
TLB::TLB(const Params *p)
: BaseTLB(p), configAddress(0), size(p->size),
     tlb(size), lruSeq(0), m5opRange(p->system->m5opRange())
{
    ...
    // L2 TLB Initialization
    if (p->l2_size >= 0) {
        is_l2 = true;
        l2_size = (uint32_t)p->l2_size;
        l2_assoc = (uint32_t)p->l2_assoc;
        l2_tlb.resize(l2_size);
```

```
            offset_bits = 12;
            DPRINTF(TLB, "L2 TLB Initialization: size-%u assoc-%u\n", l2_size, l2_assoc);
        }
        ...
    }
```

[src/arch/x86/tlb.cc]

tlb.cc implements the structure of TLB. It constructs a TLB with given parameter. The variables below are also defined in **tlb.hh**.

- **is_l2:** whether this TLB is L2 TLB or not.
- **l2_size:** the size of L2 TLB.
- **l2_assoc:** the associativity of L2 TLB.
- **l2_tlb:** a buffer where entries of L2 TLB are saved in.
- **offset_bits:** the number of offset bits of given virtual address.

```
    TLB::translate(const RequestPtr &req,
            ThreadContext *tc, Translation *translation,
            Mode mode, bool &delayedResponse, bool timing)
    {
        ...
            if (!entry) {
                DPRINTF(TLB, "Handling a TLB miss for "
                        "address %#x at pc %#x.\n",
                        vaddr, tc->instAddr());
                if (mode == Read) {
                    rdMisses++;
                } else {
                    wrMisses++;
                }
                if (is_l2) {
                    l2_Accesses++;
                    DPRINTF(TLB, "TLB lookup at 2nd level.\n");
                    entry = l2_lookup(vaddr);
                }
                if (entry) {
                    DPRINTF(TLB, "TLB Hit at 2nd level.\n");
                    Process *p = tc->getProcessPtr();
                    Addr alignedVaddr = p->pTable->pageAlign(vaddr);
                    insert(alignedVaddr, *entry);
                }
                else {
                    DPRINTF(TLB, "Final: TLB Miss\n");
                    l2_Misses++;

                    Addr alignedVaddr = p->pTable->pageAlign(vaddr);
                    DPRINTF(TLB, "Mapping %#x to %#x\n", alignedVaddr,
                                pte->paddr);
                    TlbEntry new_entry = TlbEntry(
                                   p->pTable->pid(), alignedVaddr, pte->paddr,
                                   pte->flags & EmulationPageTable::Uncacheable,
                                   pte->flags & EmulationPageTable::ReadOnly);
                  if (is_l2)
                      entry = l2_insert(alignedVaddr, new_entry);
                  entry = insert(alignedVaddr, new_entry);
                }
        ...
        return finalizePhysical(req, tc, mode);
    }
```

[src/arch/x86/tlb.cc]

This is another part of **tlb.cc** that manages TLB translation. If the processor is unable to find a required entry in the upper level of TLB (L1 TLB), it then looks for **l2_tlb** in the code.

Next, if the entry is found in **l2_tlb**, it is also inserted to L1 TLB, following the inclusive policy. However, if the entry is not found, page table lookup begins. If the entry is found in the page table, the contents are also inserted into L2 TLB and to L1 TLB. When the entry is neither found in the page table, the page fault function is invoked.

If the entry is finally obtained, it is passed to the translation process.

```
TlbEntry*
TLB::l2_lookup(Addr va, bool update_lru)
{
    if (l2_size == 0) return NULL;
    uint32_t set = (va >> offset_bits) % (l2_size/l2_assoc);
    uint32_t start = set * l2_assoc;
    uint32_t end = (set+1) * (l2_assoc);
    for (uint32_t i = start; i < end; i++) {
        DPRINTF(TLB, "l2_tlb[%u]:%#x, %#x\n", i, l2_tlb[i].vaddr >> offset_bits, va >> offset_bits);
        if ((l2_tlb[i].vaddr >> offset_bits) == (va >> offset_bits)) {
            l2_tlb[i].lruSeq = update_lru? nextSeq() : l2_tlb[i].lruSeq;
            DPRINTF(TLB, "2LTLB Hit\n");
            return &l2_tlb[i];
        }
    }
    DPRINTF(TLB, "2LTLB Miss\n");
    return NULL;
}
```

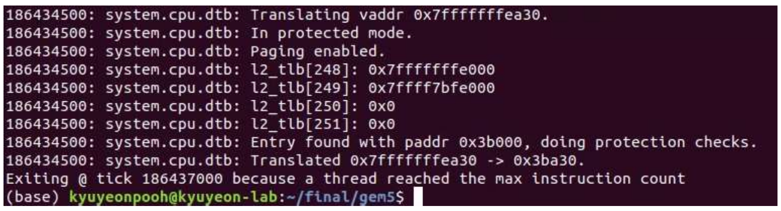
[src/arch/x86/tlb.cc]

This is another part of **l2_tlb** for lookup. As L2 TLB here is fully associative, iterating through L2 TLB with the number of L2 associativity is enough. Now, we should know where given virtual address belongs to which set. Because set is determined by virtual page number, we should shift virtual address to the right by **offset_bits**. The set is finally the remain after dividing **l2_size** by **l2_assoc**.
While searching for the entry, if virtual page number of **l2_tlb[i]** and that of virtual address are same, it returns i-th element of **l2_tlb**, otherwise **NULL**.

```
TlbEntry *
TLB::l2_insert(Addr alignedVaddr, const TlbEntry &entry)
{
    // If somebody beat us to it, just use that existing entry.
    if (l2_size == 0) return NULL;
    uint32_t set = (alignedVaddr >> offset_bits) % (l2_size/l2_assoc);
    uint32_t start = set * l2_assoc;
    uint32_t end = (set+1) * l2_assoc;
    uint32_t loc = start;
    for (uint32_t i = start; i < end; i++) {
        if (l2_tlb[i].lruSeq < l2_tlb[loc].lruSeq) {
            loc = i;
        }
    }
    TlbEntry *newEntry = &l2_tlb[loc];
    *newEntry = entry;
    newEntry->lruSeq = nextSeq();
    newEntry->vaddr = alignedVaddr;
    return newEntry;
}
```

[src/arch/x86/tlb.cc]

This is the other final part of the code in **tlb.cc** that inserts new entry into L2 TLB. It first computes the set number that corresponds to **alignedVaddr**. Then, it looks for victim entry that has lowest **lruSeq**, then inserts the given entry. **lruSeq** of empty **l2_tlb** is set to 0. Otherwise if set has empty entry, the given entry will be added to that space.

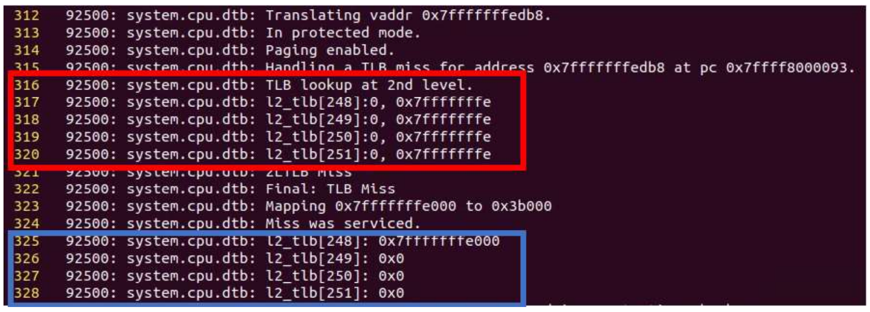


```
186434500: system.cpu.dtb: Translating vaddr 0x7fffffffea30.
186434500: system.cpu.dtb: In protected mode.
186434500: system.cpu.dtb: Paging enabled.
186434500: system.cpu.dtb: l2_tlb[248]: 0x7ffffffe000
186434500: system.cpu.dtb: l2_tlb[249]: 0x7ffff7bfe000
186434500: system.cpu.dtb: l2_tlb[250]: 0x0
186434500: system.cpu.dtb: l2_tlb[251]: 0x0
186434500: system.cpu.dtb: Entry found with paddr 0x3b000, doing protection checks.
186434500: system.cpu.dtb: Translated 0x7fffffffea30 -> 0x3ba30.
Exiting @ tick 186437000 because a thread reached the max instruction count
(base) kyuyeonpooh@kyuyeon-lab:~/final/gem5$
```

This picture is the part of printed output of **run_part1.sh**.

The command is:

```
build/X86/gem5.opt --debug-flags=TLB configs/example/speccpu2006.py \
   --cpu-type=MinorCPU --benchmark=bzip2 -I 100000 --caches --l2cache \
   --l2dtb --l2dtb_size=256 --l2dtb_assoc=4
```



```
92500: system.cpu.dtb: Translating vaddr 0x7fffffffedb8.
92500: system.cpu.dtb: In protected mode.
92500: system.cpu.dtb: Paging enabled.
92500: system.cpu.dtb: Handling a TLB miss for address 0x7fffffffedb8 at pc 0x7ffff8000093.
92500: system.cpu.dtb: TLB lookup at 2nd level.
92500: system.cpu.dtb: l2_tlb[248]:0, 0x7ffffffe
92500: system.cpu.dtb: l2_tlb[249]:0, 0x7ffffffe
92500: system.cpu.dtb: l2_tlb[250]:0, 0x7ffffffe
92500: system.cpu.dtb: l2_tlb[251]:0, 0x7ffffffe
92500: system.cpu.dtb: 2LTLB Miss
92500: system.cpu.dtb: Final: TLB Miss
92500: system.cpu.dtb: Mapping 0x7ffffffe000 to 0x3b000
92500: system.cpu.dtb: Miss was serviced.
92500: system.cpu.dtb: l2_tlb[248]: 0x7ffffffe000
92500: system.cpu.dtb: l2_tlb[249]: 0x0
92500: system.cpu.dtb: l2_tlb[250]: 0x0
92500: system.cpu.dtb: l2_tlb[251]: 0x0
```

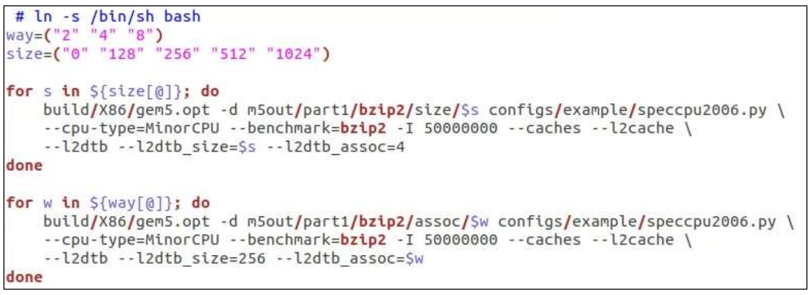
The red box is about L2 TLB lookup. Let me see the logs.

```
l2_tlb[i]: a, b
```

Here, **a** means the virtual page number of **l2_tlb[i]**, and **b** means the virtual page number of request virtual address. In the blue box, there is no corresponding entry for **l2_tlb[248:251]**, new entry is inserted to L2 TLB for address **0x7fffffe000**.

B. Results of experiments

The settings of experiment are as follow:



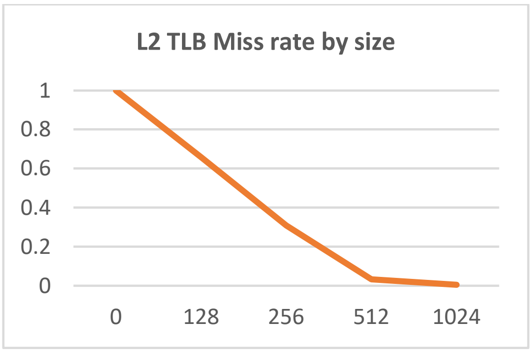
```
 # ln -s /bin/sh bash
way=("2" "4" "8")
size=("0" "128" "256" "512" "1024")

for s in ${size[@]}; do
    build/X86/gem5.opt -d m5out/part1/bzip2/size/$s configs/example/speccpu2006.py \
    --cpu-type=MinorCPU --benchmark=bzip2 -I 50000000 --caches --l2cache \
    --l2dtb --l2dtb_size=$s --l2dtb_assoc=4
done

for w in ${way[@]}; do
    build/X86/gem5.opt -d m5out/part1/bzip2/assoc/$w configs/example/speccpu2006.py \
    --cpu-type=MinorCPU --benchmark=bzip2 -I 50000000 --caches --l2cache \
    --l2dtb --l2dtb_size=256 --l2dtb_assoc=$w
done
```

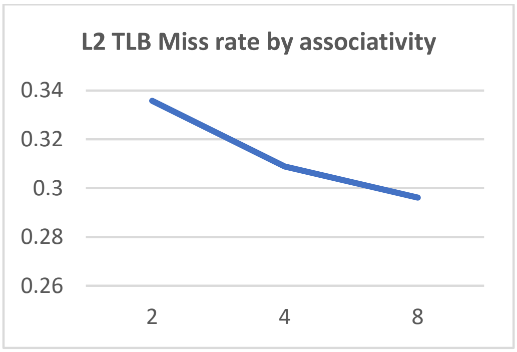
run_experiments1.sh

- CPU type: MinorCPU
- Max instruction count: 50M
- Change associativity and TLB size

Size / Assoc.	IPC	L1 TLB Miss	L2 TLB Access	L2 TLB Miss
[Size] 0	0.3955	606393	606393	606393
128	0.3955	606445	606445	399972
256	0.3955	606445	606445	187310
512	0.3955	606445	606445	20549
1024	0.3955	606441	606441	3653
[Assoc.] 2	0.3955	606445	606445	203585
4	0.3955	606445	606445	187310
8	0.3955	606445	606445	179584

Statistics of run with bzip2 benchmark





There is no change in IPC, all similar to 0.396, but that is due to no implementation of L2 miss latency. The configuration of L1 TLB is fixed, so the number of L1 TLB miss is also determined. You can check that number of L1 TLB miss and L2 TLB access is similar, about 606k.

As expected as the number of associativity in L2 TLB grows, the number of L2 TLB miss decreases. Similarly, the number of L2 TLB miss decreases as the number of entries increase.

There is about linear performance gain in terms of L2 TLB miss rate by increasing size or associativity of L2 TLB. For increasing the number of entries, the change is significant until 512. It means there may be no need of increasing the size of L2 TLB beyond 512. Increasing the associativity is still significant to performance gain while increasing the associativity from 2 to 8.

Part 2. Static Cache Partitioning

A. Explanation of implementation

```
# Way-Partition Option
parser.add_option("--div_ptr", type="int", default=-1)
```

[configs/common/Options.py]

```
# Way-Partition
if options.l2cache and options.div_ptr > 0:
    system.l2.is_divided = True
    system.l2.div_ptr = options.div_ptr
```

[configs/common/CacheConfigs.py]

Implementation about initialization and variable definition is alike Part 1. I make some parameters to configure static cache partitioning, and transfer this configuration to L2 cache. In the second source code, **div_ptr** means the position to divide the caches into two groups. For example, if the total amount of cache way here is 8 and **div_ptr** is 3, CPU 0 and CPU 1 have 3 and 5 caches each. The smaller number of ways is assigned to CPU 0, and CPU otherwise.

```
CacheBlk*
BaseCache::allocateBlock(const PacketPtr pkt, PacketList &writebacks)
{
   ...
    // Find replacement victim
    std::vector<CacheBlk*> evict_blks;
    int contextId = pkt->req->hasContextId() ? pkt->req->contextId(): -1;
    CacheBlk *victim = tags->findVictim(addr, is_secure, blk_size_bits, evict_blks, contextId, divPtr);
   ...
}
```

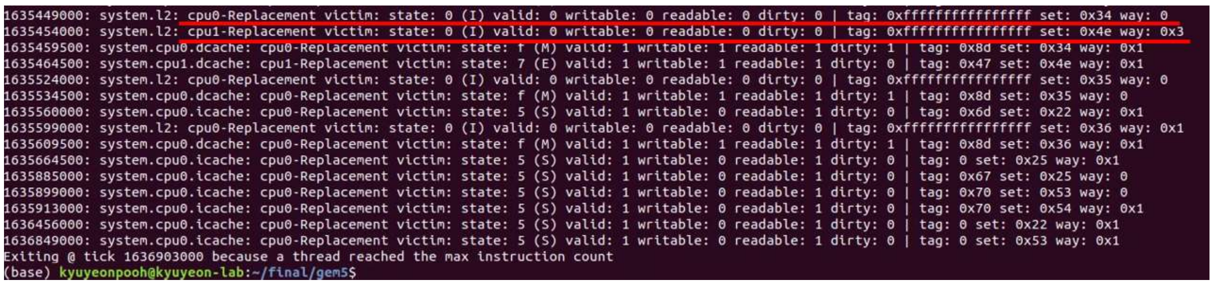
[src/mem/cache/base.cc]

BaseCache class allocates new cache block via **allocateBlock** method. Here, it finds the position where the entry is going to be inserted by **findVictim** function. By additionally passing **contextId** (the CPU ID) and **divPtr**, we can implement way-partitioning mechanism by stating the way to be excluded in that function.

```
CacheBlk* findVictim(Addr addr, const bool is_secure,
                     const std::size_t size,
                     std::vector<CacheBlk*>& evict_blks,
                     int context_id, int div_ptr) override
{
    // Get possible entries to be victimized
    std::vector<ReplaceableEntry*> entries =
        indexingPolicy->getPossibleEntries(addr);
    std::vector<ReplaceableEntry*>::iterator e = entries.begin();
    if (div_ptr >= 0 && context_id >= 0) {
        for (e = entries.begin(); e != entries.end(); e++){
            int way = (*e)->getWay();
            if ((context_id==0 && way>=div_ptr) ||
                (context_id==1 && way<div_ptr)) {
                entries.erase(e--);
            }
        }
    }
    // Choose replacement victim from replacement candidates
    CacheBlk* victim = static_cast<CacheBlk*>(replacementPolicy->getVictim(
                            entries));
    // There is only one eviction for this replacement
    evict_blks.push_back(victim);
    return victim;
}
```

[src/mem/tags/base_set_assoc.hh]

The function **findVictim** finds all possible entries via **getPossibleEntries** method, and choose a victim among them. Then, among the entry candidates, entries that do not satisfy the cache partitioning configuration are excluded from the candidate list. In other words, if CPU ID is 0 and **div_ptr** is equal to or bigger than given way, it is excluded from the possible entries. Also if CPU ID is 1 and **div_ptr** has smaller number than the given way, that way is considered further.



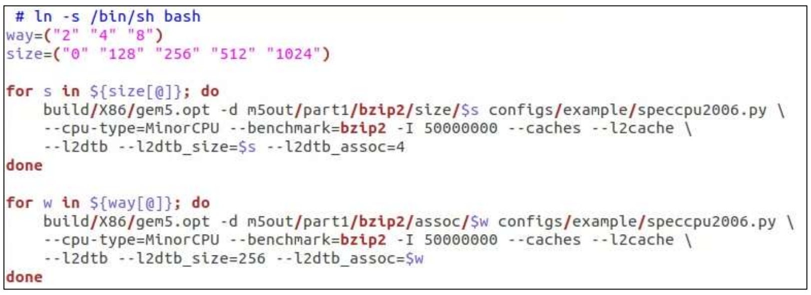
```
1635449000: system.l2: cpu0-Replacement victim: state: 0 (I) valid: 0 writable: 0 readable: 0 dirty: 0 | tag: 0xffffffffffffffff set: 0x34 way: 0
1635454000: system.l2: cpu1-Replacement victim: state: 0 (I) valid: 0 writable: 0 readable: 0 dirty: 0 | tag: 0xffffffffffffffff set: 0x4e way: 0x3
1635459500: system.cpu0.dcache: cpu0-Replacement victim: state: f (M) valid: 1 writable: 1 readable: 1 dirty: 1 | tag: 0x8d set: 0x34 way: 0x1
1635464500: system.cpu1.dcache: cpu1-Replacement victim: state: 7 (E) valid: 1 writable: 1 readable: 1 dirty: 0 | tag: 0x47 set: 0x4e way: 0x1
1635524000: system.l2: cpu0-Replacement victim: state: 0 (I) valid: 0 writable: 0 readable: 0 dirty: 0 | tag: 0xffffffffffffffff set: 0x35 way: 0
1635534500: system.cpu0.dcache: cpu0-Replacement victim: state: f (M) valid: 1 writable: 1 readable: 1 dirty: 1 | tag: 0x8d set: 0x35 way: 0
1635560000: system.cpu0.icache: cpu0-Replacement victim: state: 5 (S) valid: 1 writable: 0 readable: 1 dirty: 0 | tag: 0x6d set: 0x22 way: 0x1
1635599000: system.l2: cpu0-Replacement victim: state: 0 (I) valid: 0 writable: 0 readable: 0 dirty: 0 | tag: 0xffffffffffffffff set: 0x36 way: 0x1
1635609500: system.cpu0.dcache: cpu0-Replacement victim: state: f (M) valid: 1 writable: 1 readable: 1 dirty: 1 | tag: 0x8d set: 0x36 way: 0x1
1635664500: system.cpu0.icache: cpu0-Replacement victim: state: 5 (S) valid: 1 writable: 0 readable: 1 dirty: 0 | tag: 0 set: 0x25 way: 0x1
1635885000: system.cpu0.icache: cpu0-Replacement victim: state: 5 (S) valid: 1 writable: 0 readable: 1 dirty: 0 | tag: 0x67 set: 0x25 way: 0
1635899000: system.cpu0.icache: cpu0-Replacement victim: state: 5 (S) valid: 1 writable: 0 readable: 1 dirty: 0 | tag: 0x70 set: 0x53 way: 0
1635913000: system.cpu0.icache: cpu0-Replacement victim: state: 5 (S) valid: 1 writable: 0 readable: 1 dirty: 0 | tag: 0x70 set: 0x54 way: 0x1
1636456000: system.cpu0.icache: cpu0-Replacement victim: state: 5 (S) valid: 1 writable: 0 readable: 1 dirty: 0 | tag: 0 set: 0x22 way: 0x1
1636849000: system.cpu0.icache: cpu0-Replacement victim: state: 5 (S) valid: 1 writable: 0 readable: 1 dirty: 0 | tag: 0 set: 0x53 way: 0x1
Exiting @ tick 1636903000 because a thread reached the max instruction count
(base) kyuyeonpooh@kyuyeon-lab:~/final/gem5$
```

The picture above is part of the output by running **run_part2.sh**. It gives 3 ways to CPU 0 (way 0 to 2) and remaining ways (way 3 to 7) to CPU 1 by setting **div_ptr** to 3. The underlined log implies that CPU 0 and CPU 1 find victim only in their own partition. The command of **run_part2.sh** is as follows:

```
build/X86/gem5.opt --debug-flags=CacheRepl configs/example/speccpu2006.py --cpu-type=MinorCPU \
--benchmark=mcf,bzip2 -n 2 -I 1000000 --caches --l2cache --div_ptr=3
```

B. Results of experiments

The settings of experiment are as follow:

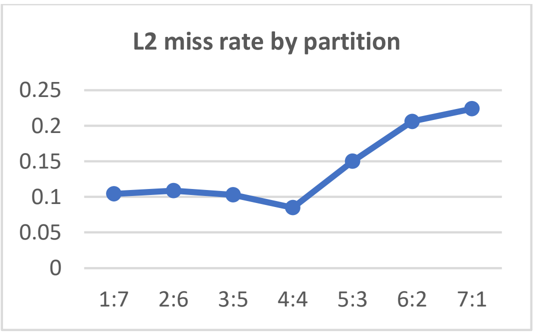


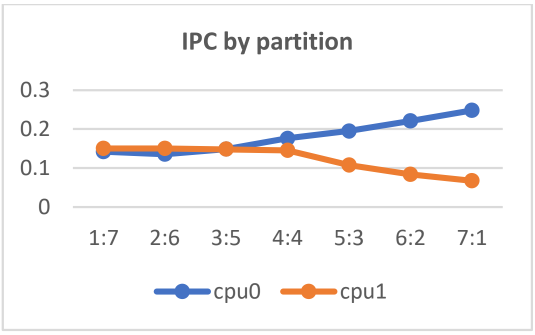
```
 # ln -s /bin/sh bash
way=("2" "4" "8")
size=("0" "128" "256" "512" "1024")

for s in ${size[@]}; do
    build/X86/gem5.opt -d m5out/part1/bzip2/size/$s configs/example/speccpu2006.py \
    --cpu-type=MinorCPU --benchmark=bzip2 -I 50000000 --caches --l2cache \
    --l2dtb --l2dtb_size=$s --l2dtb_assoc=4
done

for w in ${way[@]}; do
    build/X86/gem5.opt -d m5out/part1/bzip2/assoc/$w configs/example/speccpu2006.py \
    --cpu-type=MinorCPU --benchmark=bzip2 -I 50000000 --caches --l2cache \
    --l2dtb --l2dtb_size=256 --l2dtb_assoc=$w
done
```

run_experiments2.sh

- CPU type: MinorCPU
- Max instruction count: 10M
- L1 I cache: 1KB 8-way, L1 D cache: 1KB 8-way
- L2 cache: 64KB 8-way with 64KB block size.

B-1. Running libquantum and mcf with variety of cache portions.



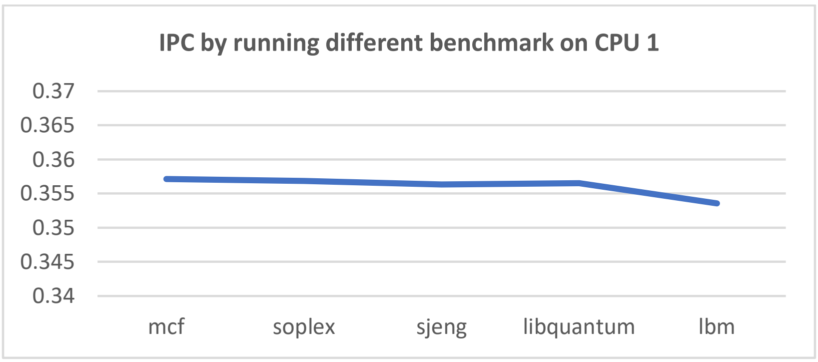


The first experiment is held with two benchmarks: **libquantum** and **mcf**. The results involve 7 different runs having partitioning ranging from 1:7 to 7:1. The left portion is given to CPU 0 and the other for CPU 1. Among 7 partition settings, 4:4 shows the best, in terms of both L2 miss rate and sum of IPC.

The other we can know from this result is the sensitivity of each benchmark along the cache size. Although relatively small amount of cache is given to **libquantum** (1:7, 2:6, 3:5), this small amount does not affect the performance of **libquantum** much. There is no dramatical change of L2 miss rate and IPC until partition 4:4. However, **mcf** benchmark shows significant performance degradation when small amount of cache is given.

Thus, although **libquantum** and **mcf** show optimal performance by statically dividing the cache equally, it seems that **mcf** requires enough cache, and more sensitive to lack amount of cache.

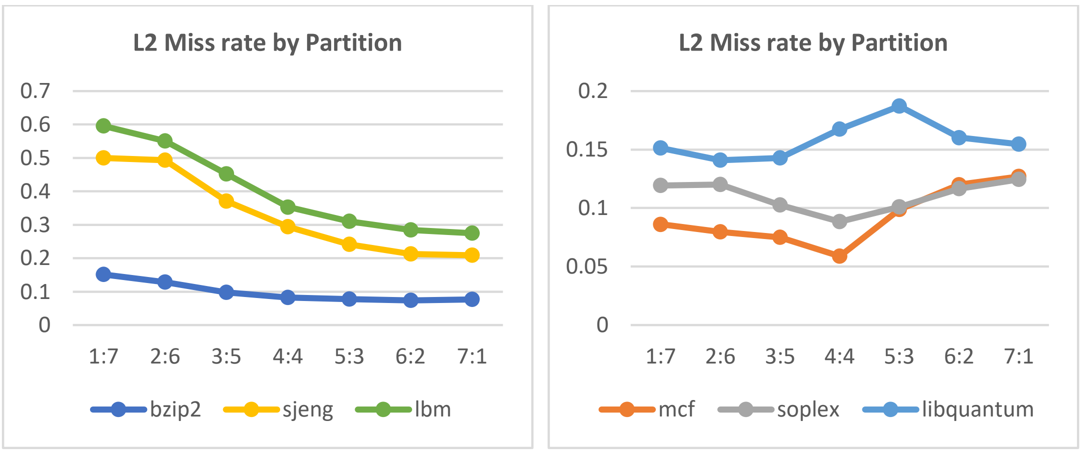
B-2. No change in partitioning, but benchmark change in only CPU 1.



In this experiment, CPU 0 always runs **bzip2** benchmark, and cache partition of fixed to 4:4, equal amount for each core. The benchmark in CPU 1 is changed for different run. This experiment is intended to check if the performance of CPU 0 is conserved by applying cache partitioning.

As shown in the figure the IPC of CPU 0 is not influenced along all benchmark: about 0.35. This implicitly tells that the performance of CPU 0 is somewhat guaranteed by reserved amount of cache, so that CPU 0 is not hindered by cache use of CPU1. There should be ablative experiment of not applying the partitioning to strictly prove this argument, but it is obvious that CPU 0 has reserved amount of cache and not being influenced by other benchmarks which may use large amount of cache.

B-3. Fixed benchmark in CPU 0, and running with various kinds of partition and benchmark in CPU 1.



In this experiment, the benchmark in CPU 0 is always fixed to **soplex**. Then, I run 6 different benchmarks in CPU 1: **bzip2, sjeng, lbm, mcf, soplex** (equal benchmark)**, libquantum**. The experiment also involves run having different cache partitioning, ranging from 1:7 to 7:1.

The figures above depict the overall L2 miss rate from both CPU 0 and 1. The figure on the left implies that **bzip, sjeng,** and **lbm** is relatively less sensitive to amount of cache. Although the amount of cache assigned to CPU 1 gets smaller, L2 miss rate decreases. This means **soplex** in CPU 0 made most of the cache misses when it has small amount of cache, and this problem is alleviated by allocating larger amount for CPU 0. The best performance is shown by partitioning with 7:1.

On the other hand, benchmarks in the right figure implies that **mcf, soplex, libquantum** has equal sensitivity or even more sensitive to cache amount. The best result is shown in 2:6 or 4:4 partition. In 1:7, it has smaller overall L2 cache misses compared to the case of the left figure. This means **mcf** and **libquantum** has more instructions that occur cache hit by having relatively large amount of cache.