Lab3 Report

Methodology

Implementation and Hardware cost

- PLRU
  - implementation
    - Maintain a private member of class `PlruPolicy`:

      ```
      private:
          bool **PlruTree;
      ```

      `PlruTree` is a table that records the PLRU trees of all sets. Each set has its own PLRU tree. I use a `bool` array to represent this complete BST.

    - `locateEvictionWay`

      Enter this method when there is a cache miss. Traverse the PLUR tree of the given set, and get the result position.

    - `updatCacheSetReplFields`

      Traverse the PLUR tree of the given set, and reverse the path (change 1 to 0 and vice versa).

  - hardware cost

    For each cache entry we need:

    - valid bit: 1 bit
    - dirty bit: 1 bit if write-back, 0 bit if write-through
    - counter: 0 bit
    - tag: (32 - `m_sets` - `m_blocks` - 2) bits
    - data: ((2^`m_blocks`) $\times$ 32) bits

    Also we need a PLRU tree for each set:

    - PLRU tree (i.e., a `bool` array): ((2^`m_ways`)-1) `bool` elements = ((2^`m_ways`)-1) bits
- LRU-LIP
  - implementation
    - `locateEvictionWay`

      Same implementation as that of LRU. If there is an invalid way in the given set, select it. Otherwise, choose the one at LRU position.

- `updateCacheSetReplFields`

  If the current transaction is a hit, then update its cache entry's counter to 0 (MRU position), and increment by one the saturate counters ([0,`ways`-1]) of all entries at this set.

  If the current transaction is a miss, then update its cache entry's counter to `ways`-1 (LRU position), and also increment by one the saturate counters of all entries at this set.

- hardware cost

  For each cache entry we need:

  - valid bit: 1 bit
  - dirty bit: 1 bit if write-back, 0 bit if write-through
  - counter: `m_ways` bits
  - tag: (32 - `m_sets` - `m_blocks` - 2) bits
  - data: ((2^`m_blocks`) $\times$ 32) bits

- DIP

  - implementation

    - Maintain a private member of class `DipPolicy`:

      ```
      private:
          int PSEL = 0;
      ```

      `PSEL` is 2-bit saturate counter. Let set `s0` use LRU policy and `s1` use LRU-LIP policy. For other sets, if `PSEL` is 0 or 1, use LRU policy, otherwise use LRU-LIP policy. To update `PSEL`, if `s0` witnesses a miss, then `PSEL` + 1, and if `s1` witnesses a miss, then `PSEL` - 1.

    - `locateEvictionWay`

      Enter this method when there is a miss. If we are now in set `s0` or `s1`, we should first update `PSEL`. The evicted way is the one at LRU position.

    - `updateCacheSetReplFields`

      If we are now in set `s0` or `s1`, we should use LRU or LRU-LIP policy. Otherwise, choose the policy according to `PSEL`.

  - hardware cost

    For each cache entry we need:

    - valid bit: 1 bit
    - dirty bit: 1 bit if write-back, 0 bit if write-through
    - counter: `m_ways` bits
    - tag: (32 - `m_sets` - `m_blocks` - 2) bits
    - data: ((2^`m_blocks`) $\times$ 32) bits

    We also need a 2-bit counter `PSEL` for the whole cache.

- Skewed-associative
  - hash functions (reference)

    I apply 4 hash functions f_1, f_2, f_3, f_4. Each way selects its hash function $f_{way \mod 4}$.

    $H(\overline{y_n y_{n-1} \cdots y_1}) = \overline{(y_n \oplus y_1) y_{n-1} \cdots y_2}$, where $\overline{y_n y_{n-1} \cdots y_1}$ is the binary representation of y.

    $f_1(A) = (H(A_1) \oplus H^{-1}(A_2)) \oplus A_2$

    $f_2(A) = (H(A_1) \oplus H^{-1}(A_2)) \oplus A_1$

    $f_3(A) = (H^{-1}(A_1) \oplus H(A_2)) \oplus A_2$

    $f_4(A) = (H^{-1}(A_1) \oplus H(A_2)) \oplus A_1$

    where address $A = \overline{A_3 A_2 A_1 A_0}$, A_0 is the block offset, A_1 is the set index, A_2 is the last $|A_1|$ bits of the tag ($|A_1|$ is the number of bits of A_1).
  - implementation

    For an address, first get its block index. Then, for all possible ways, we can use hash functions to find its corresponding sets. Suppose for address `A`, we are now at way `w` and set `s`. If entry `{s,w}` is a hit, then we only need to update all other entries' counters of `s` later using LRU. If there is no hit, we find the entry `{s,w}` whose counter is maximal among all other possible choices. We choose this entry.

Remark. When implementing replacement policies and skewed-associative cache, we should lazily ensure that all ways of a set are initialized when we are locating eviction way.

Results

Evaluation of different cache schemes

- Replacement policies

  4 sets (16 entries), 4-way associative, write-back, write-allocate, non-skewed

Design	Benchmark 1 Miss Rate	Benchmark 2 Miss Rate	Benchmark 1 Total Cycles	Benchmark 2 Total Cycles
No Cache	100%	100%	2119702	25521510
Random Replacement	31.8%	22.94%	1403669	13038657
LRU Replacement	29.79%	17.36%	1382619	12134142
PLRU Replacement	29.72%	17.8%	1381838	12205246

Design	Benchmark 1 Miss Rate	Benchmark 2 Miss Rate	Benchmark 1 Total Cycles	Benchmark 2 Total Cycles
LRU-LIP Replacement	34.59%	28.05%	1432997	13865961
DIP Replacement	30.4%	16.79%	1389061	12041638

Analysis:

1. LRU & PLRU: PLRU has similar miss rate as that of LRU. Hence it is worth using PLRU to simulate LRU with lower hardware cost.
2. LRU & LRU-LIP & DIP: Usually LRU has good hit rate, and LRU-LIP doesn't work well in common cases. Using DIP can dynamically choose between these two policies and get better hit rate than both LRU and LRU-LIP sometimes. We may choose more sets for set dueling, so that DIP can achieve better hit rate.
3. Random: Random policy is not so bad. Sometimes it can achieve better hit rate than some inapprorpriate policies.
4. Cache: Compared with no-cache, cache can greatly enhance performance (reduce cycle number) by reducing the number of memory accesses, which cause much more cycles than cache accesses.

- Cache Associativity and Capacity

  LRU, write-back, write-allocate, non-skewed

Associativity	Benchmark 1 Miss Rate	Benchmark 2 Miss Rate	Benchmark 1 Total Cycles	Benchmark 2 Total Cycles
32-entry direct-mapped	20.53%	27.08%	1285356	13708438
32-entry 8-way-associative	13.49%	9.21%	1211519	10813467
32-entry fully-associative	13.43%	9.06%	1210845	10789750

Capacity	Benchmark 1 Miss Rate	Benchmark 2 Miss Rate	Benchmark 1 Total Cycles	Benchmark 2 Total Cycles
16-entry 8-way-associative	31.26%	12.58%	1398075	11360185
32-entry 8-way-associative	13.49%	9.21%	1211519	10813467
64-entry 8-way-associative	6.99%	7.09%	1143231	10470209

- Skewed Associative

LRU, write-back, write-allocate

Capacity and Mapping	Benchmark 1 Miss Rate	Benchmark 2 Miss Rate	Benchmark 1 Total Cycles	Benchmark 2 Total Cycles
16-entry 8-way-associative	31.26%	12.58%	1398075	11360185
16-entry 8-way-skewed-associative	40.65%	38.34%	1496625	15533451
32-entry 8-way-associative	13.49%	9.21%	1211519	10813467
32-entry 8-way-skewed-associative	13.64%	9.15%	1213040	10804177

Design benchmarks

- Replacement benchmarks

  8 sets (32 entries), 4-way associative, write-through, write-allocate, non-skewed

Replacement Policy	bench_lru Miss Rate	bench_lrulip Miss Rate	bench_lru Total Cycles	bench_lrulip Total Cycles
LRU	0.09%	41.67%	862180	1012211
LRU_LIP	40.02%	16.72%	862196	1012251
DIP	0.09%	41.67%	862180	1012227

bench_lru.c

```
#define ARRAY_SIZE 64

int main() {
    int* a = (int*)malloc(sizeof(int) * ARRAY_SIZE);
    // fill the 4 ways of the particular set
    a[0] = 0;
    a[8] = 0;
    a[16] = 0;
    a[24] = 0;
    // hits for LRU (except the cold misses), misses for LRU-LIP
    for(int j=0; j<10000; j++) {
        a[32] = 0;
        a[40] = 0;
        a[48] = 0;
        a[56] = 0;
    }
        return 0;
}
```

- Design and analysis

  We choose a certain set (to which `a[0,8,16,24,32,40,48,56]` are mapped). We focus on the cache hits/misses pattern inside the iterations.

  - For LRU policy, except the 4 misses at the first iteration, all other data accesses are hits. Miss rate $\to 0$.
  - For LRU-LIP policy, since `a[32,40,48,56]` are accessed once every iteration and are always put at the LRU position, 4 accesses are all misses in every iteration. Taking `j`'s hits into consideration, miss rate $\to 4/5$.

`bench_lrulip.cpp`

```
#define ARRAY_SIZE 64

int main() {
    int* a = (int*)malloc(sizeof(int) * ARRAY_SIZE);
    // fill the 3 ways of the particular set (fixed them at LRU position for
LRU-LIP policy)
    a[0] = 0;
    a[0] = 0;
    a[8] = 0;
    a[8] = 0;
    a[16] = 0;
    a[16] = 0;
    // hits for LRU-LIP, misses for LRU
    for(int j=0; j<10000; j++) {
        a[0] = 0;
        a[8] = 0;
        a[16] = 0;
        a[24] = 0;
        a[32] = 0;
    }
    return 0;
}
```

- Design and analysis

  We still choose a certain set (to which `a[0,8,16,24,32]` are mapped). We first access `a[0,8,16]` twice, so that they can be put at the MRU position by LRU-LIP policy. We focus on the iterations.

  - For LRU policy, thrashing occurs and all 5 data accesses in an iteration are misses (except for the first iteration), hence the miss rate $\to 5/6$ if we take `j`'s hits into consideration.
  - For LRU-LIP policy, the only misses occur between `a[24,32]`. Hence the miss rate $\to 1/3$ if we take `j`'s hits into consideration.

Remark.

1. DIP policy doesn't adpat well because the tested sets are `s0` and `s1`, but for these two programs, the set `a[0,8,16,24,32,40,48,56]` mapped to is not `s0` nor `s1`. Hence the dynamic policy doesn't work, and DIP sticks to LRU policy.
2. Write-through policy eliminates the effect of a lower/higher miss rate.

- Writehit benchmarks

  8 sets (32 entries), 4-way associative, LRU, write-allocate, non-skewed

Write Hit Policy	bench_through Miss Rate	bench_back Miss Rate	bench_through Total Cycles	bench_back Total Cycles
Write-back	20.07%	0.09%	561771	220823
Write-through	20.07%	0.09%	690115	380967

`bench_writehit_through.c`

```
#define ARRAY_SIZE 16

int main () {
  int* a = (int*)malloc(sizeof(int) * ARRAY_SIZE * 1000);
  for(int j=0; j<ARRAY_SIZE * 1000; j++) {
      a[j] = 0;
  }
  return 0;
}
```

  - Design and analysis

    The best case for write-through policy is when all writes will not be re-written later.

    - For write-back, almost all writes are misses and every data is written to memory when accessed.
    - For write-through, almost all writes are misses and every data is written to memory when evicted.

    The difference between cycle numbers is the time of write-through policy writing `j` to memory every iteration while write-back policy does not.

`bench_writehit_back.c`

```
int main() {
  int a = 0;
  for(int j=0; j<10000; j++) {
      a = j + 1;
  }
}
```

- Design and anlaysis

  The best case for write-back policy is when all writes will be re-written soon.

  - For write-back, every access to `a` is a hit and write-back is applied. Hence there is almost no memory access latency.
  - For write-through, every write to `a` is flushed down to memory, which is responsible for the cycle number gap.

Remark.

1. In terms of cycle number, write-back policy is always better than write-through policy. The advantages of write-through policy lie in its simpler implementation (no need for dirty bits). And if there is a write buffer to hide memory latency, the edge of write-back policy over write-through policy can be smaller.

- Writemiss benchmarks

  8 sets (32 entries), 4-way associative, LRU, write-through, non-skewed

Write Miss Policy	bench_allocate Miss Rate	bench_noallocate Miss Rate	bench_allocate Total Cycles	bench_noallocate Total Cycles
Write-allocate	0.09%	20.07%	380967	690115
No-write-allocate	25.11%	20.1%	381119	690275

`bench_writemiss_allocate.c`

```
int main() {
  int a = 0;
  for(int j=0; j<10000; j++) {
      a = j + 1;
  }
}
```

- Design and analysis

  The best case for write-allocate policy is the missed block will be re-written later.

  - For write-allocate, almost all writes are misses, hence the miss rate $\rightarrow 0$.
  - For no-write-allocate, all writes to `a` are misses since `a` is not allocated to the cache. The relatively low miss rate is due to the write hits to `j`.

`bench_writemiss_noallocate.c`

```
#define ARRAY_SIZE 16
```

```
int main () {
  int* a = (int*)malloc(sizeof(int) * ARRAY_SIZE * 1000);
  for(int j=0; j<ARRAY_SIZE * 1000; j++) {
      a[j] = 0;
  }
  return 0;
}
```

- Design and anlaysis

  The best case for no-write-allocate policy is the missed blocks will never be re-written later.

  - For write-allocate, every time `a[j]` need to be moved to cache, despite the fact that every `a[j]` is a miss and is only accessed once.
  - For no-write-allocate, `a[j]` is never moved to cache.

Remark.

1. When using benchmark `bench_writemiss_allocate.c`, the difference between write-allocate and no-write-allocate in terms of cycle number is eliminated by the write-through policy.
2. There seems no difference between write-allocate and no-write-allocate in terms of miss rate or cycle number when we use benchmark `bench_writemiss_noallocate.c`. It is natural that write-allocate and no-write-allocate should have similar miss rates. However, the gap in cycle number is eliminated since Ripes doesn't take into consideration the bandwidth latency to fetch a block from memory to cache. Hence no-write-allocate still has an edge over write-allocate when considering memory bandwidth.

Question Answering

1. *Do you think deep learning, or other machine learning techniques, can be applied to cache replacement ? What are the advantages and possible difficulties ?*

   Answer. It is possible for deep learning to be applied to cache replacement, since "it provides the ability to learn non-linear relationships at multiple levels of abstraction"(reference), thus generates better prediction than tables or other simpler prediction techniques. So chances are that deep learning can enhance cache hit rate. The disadvantage is the difficulty to implement deep learning in hardware level. Deep learning takes huge resources (e.g., time) and requires the targets not to change with time, but usually hardware cannot provide resources that much and programs always vary from each other greatly(reference).

2. *Do you think non-blocking cache is useful in your processor ? How much is the cost and benifit ?*

   Answer. I think non-blocking cache is useful in my 5-stage processor. For the case in which there are two seqeuntial data accesses, without a non-blocking cache, the second access must wait until the first access finishes (can be 2 cycles if hit and 10 cycles if miss), and if the first access is a miss, multiple cycles are wasted during which both the CPU and the cache are idling. If we apply a non-blocking cache, CPU will stall only when the data are needed, that is, the CPU and cache will not idle when a data need to be fetched from memory. The memory latency can be hidden. Thus the benefit is that performance is improved. The cost is the hardware cost to keep track of the data accesses that have not

yet succeeded: the target data is not in the cache and has not been fetched from memory to CPU. We can use MSHR to handle this, which increase power, energy and complexity.

3. *Do you think software prefetching is useful in your processor ? How much is the cost and benefit ?*

   Answer. I think this is not useful in my 5-stage processor, since prefetch instruction will cost 10 cycles stall (memory access latency) and does not reduce cycle number. If software prefetching is applied to an OoO processor, or the memory access latency is hidden by other techniques (i.e., prefetch instruction doesn't cause too much extra cycles), then it has the benefit to reduce miss rate (if prediction algorithm works well). However, this need a more complex compiler design. Also, for different architecture, the positions to insert prefetch instructions also differ, thus software prefetching is not quite portable.