Lab 3 Documentation

Methodology

Replacement Policy Implementation

PLRU

We implement the **MRU** policy, maintaining a bit for each entry. The hardware cost of the policies is $30 - BlockBits - SetBits$ bits for a tag, 1 dirty bit, 1 valid bit, and 1 MRU bit per entry. Also one needs to maintain $SetNum$ of the set index, each with $SetBits$ bits. Also, there are $32 \times EntryNum \times BlockNum$ data bits

Hence in total, we need hardware cost of

$$\begin{gathered}(33 - BlockBits - SetBits) \times EntryNum \\ +SetNum \times SetBits \\ +32 \times EntryNum \times BlockNum\end{gathered}$$

bits.

LRULIP

We use the same eviction policy as LRU and modify the update policy to behave differently if $ishit = False$, if this happens, we will further distinguish whether the newly inserted entry is invalid before. The hardware cost is the same as LRU, which is

$$\begin{gathered}(32 - BlockBits - SetBits + WayBit) \times EntryNum \\ +SetNum \times SetBits \\ +32 \times EntryNum \times BlockNum\end{gathered}$$

DIP

We maintain a global bit to indicate we should currently use LIP or LRU policies and four unsigned ints to indicate the number of all access and all hits to the first and second sets, which always use LRU or LIP. Finally, we have a global counter to indicate whether it's time to do a policy transition. To save hardware costs, the LRU/LIP counter for each entry is shared when a policy transfer happens. Hence the hardware cost is only slightly over LRU, which is

$$\begin{gathered}(32 - BlockBits - SetBits + WayBit) \times EntryNum \\ +SetNum \times SetBits \\ +32 \times EntryNum \times BlockNum \\ +5 \times 32 + 1\end{gathered}$$

bits.

Skew Associative Implementation

Our Implementation is as followed,

```
void CacheSim::analyzeCacheAccessSkewedCache(CacheTransaction& transaction) {
    if (this->m_type == CacheType::InstrCache) {
        return analyzeCacheAccess(transaction);
    }
    // ---------------------Part 3. TODO ------------------------------
    // Implement the skewed-associative cache
    unsigned baseset = getSetIdx(transaction.address);
    transaction.index.block = getBlockIdx(transaction.address);
    transaction.isHit = false;
    unsigned way_number = getWays();
    // check whether there is a hit
    for(unsigned k = 0; k < way_number; k++){
        unsigned possibleset = skewhash(transaction.address,k); // naive hash
        if(m_cacheSets.count(possibleset) == 0){
            m_cacheSets[possibleset];
            for(unsigned j = 0; j < way_number; j++){
                m_cacheSets[possibleset][j];
            }
        }
        if((m_cacheSets[possibleset][k].tag == getTag(transaction.address)) &&
m_cacheSets[possibleset][k].valid){
            transaction.index.way = k;
            transaction.index.set = possibleset;
            transaction.isHit = true;
            break;
        }
    }
    if (!transaction.isHit){
        // if there haven't been any hit
        // check if there is invalid way, notice here all possibleset has been init
        bool hasasign = false;
        for(unsigned k = 0; k < way_number; k++){
            unsigned possibleset = skewhash(transaction.address,k); // naive hash
            if(! m_cacheSets[possibleset][k].valid){
                transaction.index.way = k;
                transaction.index.set = possibleset;
                hasasign = true;
                break;
            }
        }
        // if haven't assign, all the possible ways are valid, have to choose one to evict
        if(! hasasign){
            unsigned max_counter = 0;
            unsigned victim = 0;
            for(unsigned k = 0; k < way_number; k++){
                unsigned possibleset = skewhash(transaction.address,k); // naive hash
                max_counter = (max_counter > m_cacheSets[possibleset][k].counter) ? max_counter
:m_cacheSets[possibleset][k].counter;
                if(max_counter == m_cacheSets[possibleset][k].counter){
                    victim = k;
                }
            }
            transaction.index.way = victim;
            transaction.index.set = skewhash(transaction.address,victim); // naive hash
        }
    }
    return;
}
```

As instructed by the handout, we implement a skew associative cache with LRU policy. We separate the case where there is a hit, where there is an invalid entry, and where we have to choose one entry to evict using LRU policy.

Our skew hash function mainly references the paper for the basic hash function H and repeatedly applies it to calculate the combination. Mathematically, we use $2 \times SetBits$ in the address, notes as $A_0\bar{}A_1$, we have $h_k(A_0\bar{}A_1) = H^{(k)}(A_0) \oplus A_1$ where $H(y_n, \ldots, y_1) = (y_n \oplus y_1, y_n, \ldots, y_1)$ as in the paper.

Results

Replacement Policies

16-entry, 4-way associative cache, 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	1	1	2119802	1032796
Random Replacement	0.3186	0.4658	1404488	745671
LRU Replacement	0.2976	**0.4291**	1382493	**725973**
PLRU Replacement	0.2969	0.4396	1381653	731569
LRU-LIP Replacement	0.3041	0.4576	1389212	741263
DIP Replacement	**0.2964**	0.441	**1381103**	732351

We can see that the total cycle counts are always positively related to the miss rates, where the no-cache and random are always the worst. We also notice that $PLRU$ is always similar to LRU, meaning we have done a correct proxy. $LRU-LIP$ generally performs worse in these two benchmarks. DIP interestingly is slightly better than all other strategies in benchmark 1, however, we observe that the policy for all the sets except set 1 has been consistently LRU, showing that this is more of a coincidence.

Cache Associativity and Capacity

Associativity	Benchmark 1 Miss Rate	Benchmark 2 Miss Rate	Benchmark 1 Total Cycles	Benchmark 2 Total Cycles
32-entry direct-mapped	0.2076	0.3412	1287939	678693
32-entry 8-way associative	0.1351	0.338	1211748	676967
32-entry fully associative	0.2343	0.3309	1210945	673156

Capacity	Benchmark 1 Miss Rate	Benchmark 2 Miss Rate	Benchmark 1 Total Cycles	Benchmark 2 Total Cycles
16-entry 8-way associative	0.3128	0.4257	1398353	724107
32-entry 8-way associative	0.1351	0.338	1211748	676967
64-entry 8-way associative	0.0666	0.2067	1139873	606387

We observe that the hit rate consistently increases with associativity and capacity which is as expected.

Skew Associative

Capacity and Mapping	Benchmark 1 Miss Rate	Benchmark 2 Miss Rate	Benchmark 1 Total Cycles	Benchmark 2 Total Cycles
16-entry 8-way associative	0.3128	**0.4257**	1398353	**724107**
16-entry 8-way skewed associative	**0.3063**	0.4319	**1391566**	727445
32-entry 8-way associative	**0.1351**	0.338	**1211748**	676967
32-entry 8-way skewed associative	0.1365	**0.3314**	1213276	**673442**

Our skewed associative cache and the associative cache are comparable. We believe the reason why our skewed associative cache hasn't been able to consistently outperform the associative cache is due to the negative impact of maintaining a not very reasonable LRU counter within a. set, where this set is not as useful in the skewed associative cache.

LIP&LRU

We design the benchmark using the following simple code.

```
// bench_lrulip
int main(){
    int a[100];
    int sum = 0;
    for(int k = 0; k < 1000; k++){
        for(int i = 0; i < 9; i++){
            sum += a[4*i];
        }
    }
    return 0;
}
// bench_lru
int main(){
    int a[100];
    int sum = 0;
    for(int k = 0; k < 1000; k++){
        for(int i = 0; i < 7; i++){
            sum += a[4*i];
        }
    }
    return 0;
}
```

We know in class that LRU suffers from iterating an array if the iteration lasts too long, on the other hand, if all the iteration variables can fit into the cache LRU is a near-optimal policy. Hence we design two iterations with different lengths. We try to control the length to be slightly above or below the way number and find the result is as expected.

We also notice that DIP always shows performances between LRU and LRULIP in this two benchmarks.

Replacement Policy	bench_lru miss rate	bench_lrulip miss rate	bench_lru Total Cycles	bench_lrulip Total Cycles
LRU	0.001	0.1556	377029	544973
LRU-LIP	0.1311	0.1384	424981	536981
DIP	0.0877	0.1356	408981	544957

Write Hit Policy

Write-back is always better than write-through in total cycle count and two policies have the same miss rate.

The reason is that for write-allocate write-through and write-back policies, one can prove that for the same sequence of cache access, the cache is always the same. Hence the miss rate is always the same. However, for the write-back cache, one doesn't have to access the memory on every write, hence can save total cycle counts.

We use a benchmark to show our point.

```
int main(){
    int a[1000];
    for(int k = 0; k < 1000; k++){
        a[0] = k;
    }
    return 0;
}
```

Hit Policy	bench_through miss rate	bench_through total cycles
Write-back	0.0083	23812
Write-through	0.0083	39940

Write Miss Policy

```
// allocate
int main(){
    int a[1000];
    for(int k = 0; k < 1000; k++){
        a[0] = k;
    }
    return 0;
}
// no allocate
int main(){
    int a[100];
    int sum = 0;
    for(int k = 0; k < 1000; k++){
        for(int j = 0; j < 32;j++){
            a[j] = k;
        }
        for(int j = 32; j < 64;j++){
            sum += a[j];
        }
    }
    return 0;
}
```

Intuitively, the write-allocate policy is better when we will quickly revisit the previously written address. Hence we design the benchmark as above. In bench_allocate, we will always visit the written address repeatedly. In bench_noallocate, we will always read many other addresses before revisiting the written address.

Write Miss Policy	**bench_allocate miss rate**	**bench_noallocate miss rate**	**bench_allocate Total Cycles**	**bench_noallocate Total Cycles**
Write-allocate	0.0083	0.19	39940	3213014
No Write-allocate	0.261	0.1427	40100	3085286

QA

AI For Cache

After reading the paper, I believe one has to use deep learning indirectly.

So far, to hardwire a trainable AI inside a cache is still overkill, as it will be very costly and time-consuming. However, one can imagine using an AI to analyze the data flow to choose between different policies on a timely basis, much similar to our DIP. One can naturally model this question as a multi-arm bandit problem.

Non-blocking cache

For our simple five-stage processor, the non-blocking cache is not very useful. As when a cache miss happens, the pipeline is stalled. Hence there will never be multiple cache miss into the cache for the MSHRs to handle. In principle, a non-blocking cache is only useful in an OoO processor. Hence without reimplementing the processor, the cost is more hardware costs and there won't be any benefit.

Prefetching

I believe this is useful, as this can significantly utilize the empty cache and memory bus and increase the hit rate.

The cost will be much more complicated hardware implementation, including a more sophisticated ISA and avoiding stalling for the prefetching command. Also, one has to carefully handle the case where a prefetching hasn't been completed and a read to the same address is already required, either to avoid this by using compilers or adding even more hardware supports.

Bonus

It has been a very long journey to learn computer architecture but it is full of joy. The topic to analyze algorithm optimizations, code optimizations, and microarchitecture optimizations is very profound and interesting.

Algorithm Optimization

I entered Yao class with a mathematics Olympic background and for the first 16 years of my life, I have been focusing more on doing things right, instead of doing things fast. I remember the first time I learned how to sort in $O(n \log n)$ time and how to prove one can never improve over this asymptotic bound, I was much surprised and started to understand the complexity theorem, which arguably is the foundation of theoretical computer science. Though I can't claim to be a master in this field, I realized there is a significant difference between a polynomial algorithm and an exponential one. Up to this day, due to a theoretical preference, I still feel that finding the correct and well-optimized algorithm is the first step to solving a problem.

Code Optimization

Last summer, while learning *Data Structure in Real World*, I began to realize how important it is to write well-optimized code. It's stunning to see how one can incrementally improve one's code to achieve performance ten times or one hundred times faster, using exactly the same algorithm and programming language. Frankly speaking, before which I always wonder what it means to write good code and after which I started to realize what is the difference between a rookie and a professional.

Microarchitecture Optimization

This year, the inspiring news of Google's pathway structure makes me realize that the beauty of machine learning is fundamentally based on carefully designed hardware microarchitecture, and thanks to this course I took a baby step toward designing better microarchitecture. So far, this course has the most well-documented code base and clearest instruction for all the courses I have taken at Tsinghua. Big thanks to TAs and professor Gao.