ML-based Approach on

Cache Replacement

2019-2 Yonsei University Capstone Final Report

Cacher

2016163055 Seongwoo Kim

2011122011 Minjun Lee

2016121114 Hyeonjae Cho

Prof. Shin-dug Kim

Super Computing Lab.

Contents

[2019-2 CS Capstone]

Cache Replacement Algorithm

2016163055 Seongwoo Kim
2011122011 Minjun Lee
2016121114 Cho Hyeon Jae

1. Background

Cache is a small and fast memory that is equipped in the CPU. Since accessing main memory costs time consumption to the great extent, we keep the most frequently accessed data into the cache. When the processor checks through all cache for target data and nothing matched, we call this situation as ‘cache miss’ and we operate cache replacement policies. Recent studies on cache replacement policies are mainly focused on ‘LLC’ cache with machine learning based algorithm. The most traditional studies done on this area includes FIFO and LRU algorithm.

1-1. Target : What We Focus On

Our study mainly targeted developing the algorithm which helps LLC hit rate and enhance the whole system performance. Also, we decided that our algorithm should be practical that the algorithm can be utilized in reality and in online-fashion. The reason why we focused more on practicalness is that most recently studied papers are about elaborate machine learning based algorithm, but it is useless in reality. Those algorithms have drawbacks in that the algorithm itself has large overheads, so when it implemented in the cache, it downgrades overall performance ironically. Therefore, our team put LRU as the baseline to test our new algorithm.

1-2. Limitations of Traditional Approaches

Current studies can be divided into two groups : one is ‘Complex but theoretical’ and the other is ‘Simple but low performance’. Deep learning based cache replacement algorithms such as Hawkeye are involved in the former. Traditional algorithms such as FIFO, LRU and recently used algorithms like RRIP are included in the latter. As the group name indicates, deep learning based algorithms can not be used in reality because of its overheads. Simple algorithms show bad performance in reality and also differ to great amount depends on which trace file it uses. As we mentioned before, our team target is more focused on ‘practicalness’ rather than ideal performance.

2. Methodology

There were several steps before we jumped into developing our own algorithm. First, we imported lab’s code of cache simulator and adjusted it into more simple way. Our team took about two weeks to add Hawkeye, LRU, FIFO algorithm inside and tested several trace files to figure out the best working files to test our own algorithm in the future.

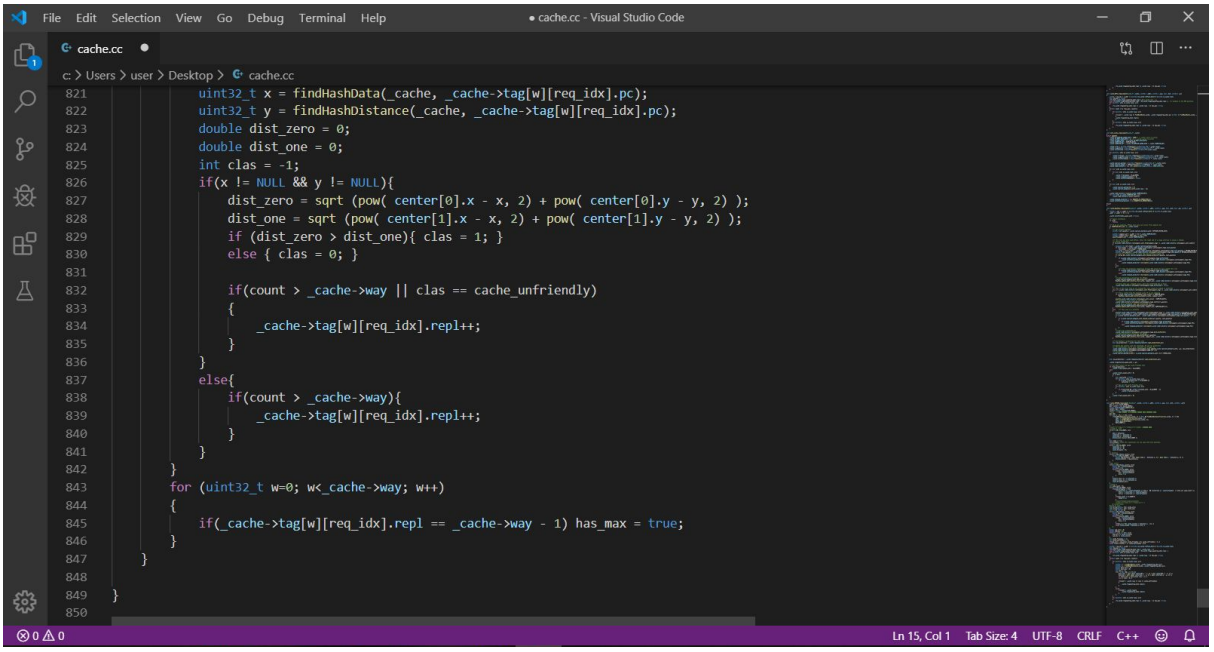


Figure. Code snippet of main .cc file of our final cache simulator code (about 850 lines)

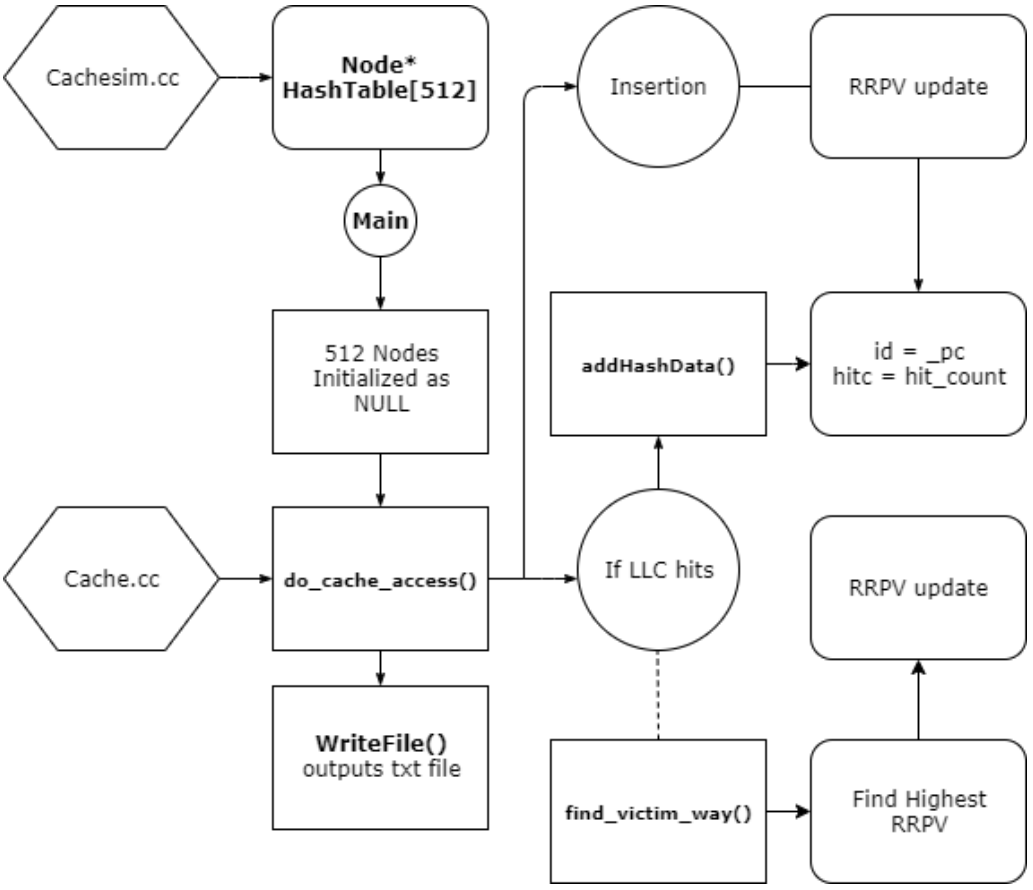


Figure. Code Overflow Chart of Main

After completion of our simple cache simulator code, our team focused to figure out proper test trace files. The supercomputing lab provided more than 20 trace files. We tested them all with each FIFO, LRU and Hawkeye and filtered them into 7 trace files. The condition we applied was that with each different algorithm, trace file should show differentiated result, which means that it sensitively reacts to the difference of replacement algorithms. The selected 7 trace files are listed as following:

mcf, soplex, leslie3d, milc, GemsFDTD, omnetpp, sphinx3

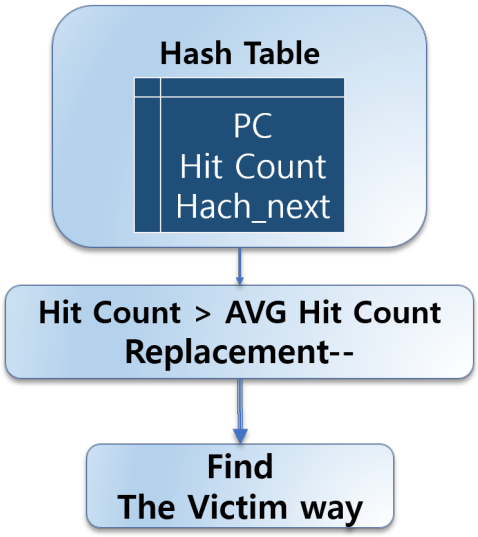
Above trace files are from different workloads and their working set size differs and denoted by following numbers (e.g. mcf-184B). These trace files usually have bigger working sets than L1 or L2 cache size, which means that they are perfect test files to test the extent of LLC hit rate improvement.

Trace File	Algorithm	# of Instruction	L1$ hit rate	L2$ hit rate	L3$ hit rate
456.hmmer-88B	FIFO	10000000000	95.823%	52.134%	83.561%
456.hmmer-88B	FIFO	1000000000	95.777%	52.063%	84.614%
456.hmmer-88B	FIFO	100000000	95.972%	53.676%	80.850%
456.hmmer-88B	FIFO	10000000	95.596%	53.994%	75.318%
456.hmmer-88B	LRU	10000000000	99.319%	25.014%	86.197%
456.hmmer-88B	LRU	1000000000	99.319%	24.959%	90.343%
456.hmmer-88B	LRU	100000000	99.321%	24.932%	36.129%
456.hmmer-88B	LRU	10000000	99.307%	24.138%	4.878%
456.hmmer-88B	HAWKEYE	10000000000	94.969%	57.323%	92.472%
456.hmmer-88B	HAWKEYE	1000000000	95.011%	57.599%	92.537%
456.hmmer-88B	HAWKEYE	100000000	96.115%	59.185%	82.557%
456.hmmer-88B	HAWKEYE	10000000	97.137%	56.108%	60.204%
403.gcc-48B	FIFO	10000000000	99.554%	45.331%	30.883%
403.gcc-48B	FIFO	1000000000	99.416%	37.022%	27.709%
403.gcc-48B	FIFO	100000000	99.735%	71.505%	44.789%
403.gcc-48B	FIFO	10000000	99.75%	79.17%	1.29%
403.gcc-48B	LRU	10000000000	99.551%	67.090%	18.866%
403.gcc-48B	LRU	1000000000	99.448%	56.566%	16.153%
403.gcc-48B	LRU	100000000	99.679%	86.789%	16.680%
403.gcc-48B	LRU	10000000	99.675%	84.353%	0.000%
403.gcc-48B	HAWKEYE	10000000000	99.412%	51.031%	49.624%
403.gcc-48B	HAWKEYE	1000000000	99.385%	40.400%	35.622%
403.gcc-48B	HAWKEYE	100000000	99.828%	74.653%	4.462%
403.gcc-48B	HAWKEYE	10000000	99.822%	71.400%	0.000%
654.roms_s-1021B	FIFO	1000000000	99.503%	34.138%	77.826%
654.roms_s-1021B	FIFO	100000000	99.906%	83.850%	18.481%
654.roms_s-1021B	FIFO	10000000	99.960%	25.092%	0.000%
654.roms_s-1021B	LRU	1000000000	99.879%	26.253%	53.943%
654.roms_s-1021B	LRU	100000000	99.979%	40.827%	0.000%
654.roms_s-1021B	LRU	10000000	99.962%	20.392%	0.000%
654.roms_s-1021B	HAWKEYE	1000000000	99.462%	42.106%	79.429%
654.roms_s-1021B	HAWKEYE	100000000	99.941%	77.672%	7.000%
654.roms_s-1021B	HAWKEYE	10000000	99.964%	16.289%	0.000%
644.nab_s-5853B	FIFO	1000000000	97.443%	44.735%	72.459%
644.nab_s-5853B	FIFO	100000000	98.148%	79.626%	86.248%
644.nab_s-5853B	FIFO	10000000	98.157%	81.848%	71.816%
644.nab_s-5853B	LRU	1000000000	99.172%	93.016%	15.886%
644.nab_s-5853B	LRU	100000000	99.249%	92.636%	6.170%
644.nab_s-5853B	LRU	10000000	98.977%	90.773%	0.079%
644.nab_s-5853B	HAWKEYE	1000000000	96.159%	48.998%	96.992%
644.nab_s-5853B	HAWKEYE	100000000	98.720%	95.160%	16.267%
644.nab_s-5853B	HAWKEYE	10000000	98.964%	90.870%	0.290%

Figure. More than a million Instructions

Finally, We developed three main algorithms : Hit count, Reuse distance, K-means algorithm. The algorithm names somehow show how we approached each replacement policies.

2-1. Hit Count Algorithm



Hit count algorithm uses hit count as the standard to compare priority to be replaced. If the input data has lower hit count than existing data, it's rrpv is set high value so that it can be replaced as much as possible and keeps cache to store more frequently used data inside. The performance it showed will be treated in the "3. Performance Analysis". The flow chart of this algorithm is given on the left of this paragraph.

Logic

```
HITC_replacement () :
Input : cache, address, way, hit,
program counter
Output : void
If cache_hit == False then set
rrpv to way-2
Else rrpv --
While ( ! find_victim_way):
   // change priorities
   hit_count > average_hitc :
      rrpv ++;
```

Measures

RRPV	0	1	Way - 1	Way – 2
Total Execution Cycle	408932415	408919247	408184719	399747523
Average IPC	0.24	0.24	0.24	0.25
L1$ Hit Rate	97.862	97.834	97.557	97.806
L2$ Hit Rate	15.559	17.012	22.037	21.557
L3$ Hit Rate	18.046	17.762	25.032	20.884

437.leslie3d-232B

Figure. Logic and Performance Result of Hit Count Algorithm

Hit count replacement algorithm changes priorities within its function after treating new inserted data and its RRPV setting. It compares existing data with average hit count value and replace rrpv values to change priorities. Through this, we could filter out less frequently used data in the cache and improve the LLC hit rate.

With our algorithm logic, we tested several trace files to figure out the best rrpv value to set when the cache miss occurs. As the figure shows, with leslie3d trace file, the optimized RRPV value is max way - 2. The first algorithm is completed.

2-2. Reuse Distance Algorithm

Reuse distance algorithm puts standard on reuse distance of data. If the data has short reuse distance, then it means that data would hit sooner than any other data. This means that the data is frequently used, so we would like to store these kinds of data inside the cache.

The overall algorithm logic resembles much with Hit Count Algorithm. This also changes the priorities of entire stored cache data when replacement function evokes. When the existing data has longer reuse distance than average, then this data priority goes up to be replaced sooner.

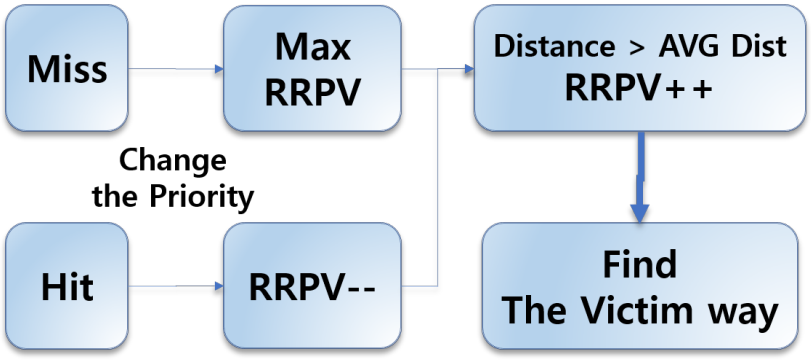


Figure. Flow chart of Reuse Distance Algorithm

Logic

```
DIST_replacement () :
Input : cache, address, way,
hit, program counter
Output : void
If cache_hit == False then set
rrpv to max_way
Else rrpv --
While ( ! find_victim_way):
   // change priorities
   distance > average_dist :
      rrpv ++;
```

Measures

RRPV	0	1	Way - 1	Way - 2
Total Execution Cycle	486731744	486023508	479119396	369014988
Average IPC	0.27	0.21	0.21	0.27
L1$ Hit Rate	94.683	94.649	94.582	94.361
L2$ Hit Rate	11.162	11.957	15.016	21.292
L3$ Hit Rate	6.425	6.496	7.121	55.134

482.sphinx3-1297B

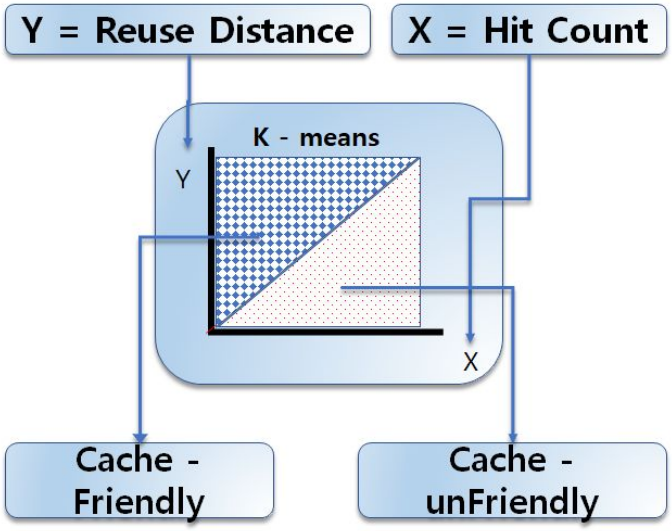
Figure. Logic and Performance Result of Reuse Distance Algorithm

Hit Count Algorithm performed best when missed data RRPV value sets to max way - 2. On the other hand, reuse distance algorithm performs better when the RRPV value is set as max

way - 1. This means that this algorithm works better when evicting cache unfriendly data as soon as possible.

Compared to Hit Count Algorithm, reuse distance worked best in four trace files among seven. It ruled out FIFO, LRU and Hit Count Algorithm in four trace files : sphinx3, milc, mcf, and soplex.

2-3. K-means Algorithm



we implemented K-means algorithm with some change. K-means algorithm need some amount of data when we run the program. However, we need to use the algorithm for every cache misses. It's very inefficient to compute every point for each execution. Therefore, we just divided data for two groups. Every time when new data with its pc value is in hash table comes, it decide which group is closer with the data. The coordinate of data is represented as (hitcount, reuse distance). If the data is closer to the group with higher hitcount value, the program will not add 1 to the rrpv value which means that the data is cache friendly. After that, we add the value of the coordinate to the near group to adjust coordinate of the group. The rest parts of the algorithm is similar with HITC replacement.

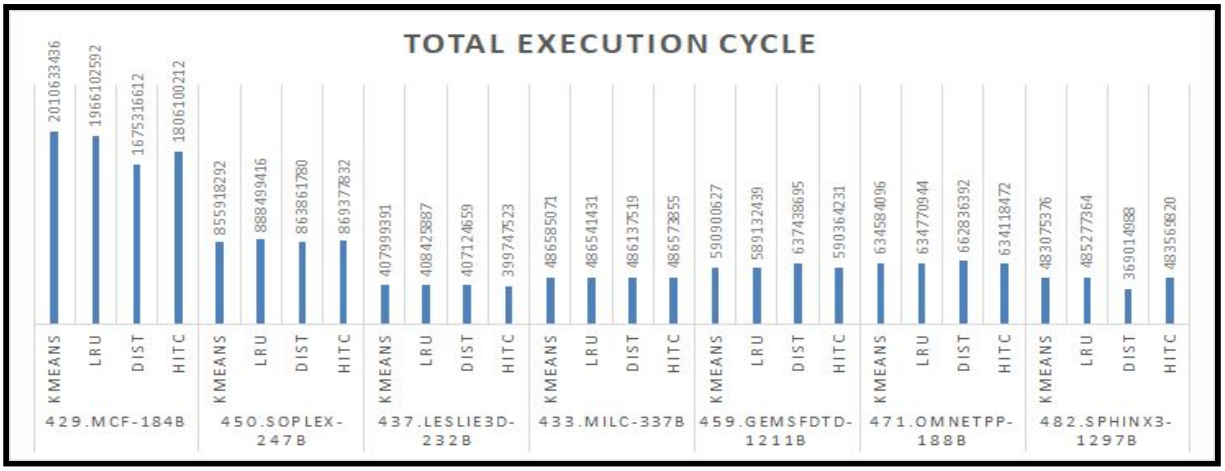
3. Performance Analysis

Our team checked the performance of our algorithm by modifying and applying the existing simulator. Simulate 7 trace files to check the overall performance and performance time first. The smaller the total execution cycle, the better the performance. Then check the IPC. IPC is an indicator of the number of commands Instructions Per Cycle, so the higher the better. Finally, the hit rate was analyzed. Especially since we targeted LLC, we compared it with L3. And we compared the geometric mean of the collected data, rather than the arithmetic mean, to see the performance increase.

3-1. Total Execution Cycle

Trace File	Algorithm	Total Execution Cycle	Trace File	Algorithm	Total Execution Cycle
429.mcf-184B	KMEANS	2010633436	459.GemsFDTD-1211B	KMEANS	590900627
	LRU	1966102592		LRU	589132439
	DIST	1675316612		DIST	637438695
	HITC	1806100212		HITC	590364231
450.soplex-247B	KMEANS	855918292	471.omnetpp-188B	KMEANS	634584096
	LRU	888499416		LRU	634770944
	DIST	863861780		DIST	662836392
	HITC	869377832		HITC	634118472
437.leslie3d-232B	KMEANS	407999391	482.sphinx3-1297B	KMEANS	483075376
	LRU	408425887		LRU	485277364
	DIST	407124659		DIST	369014988
	HITC	399747523		HITC	483569820
433.milc-337B	KMEANS	486585071			
	LRU	486541431			
	DIST	486137519			
	HITC	486573855			

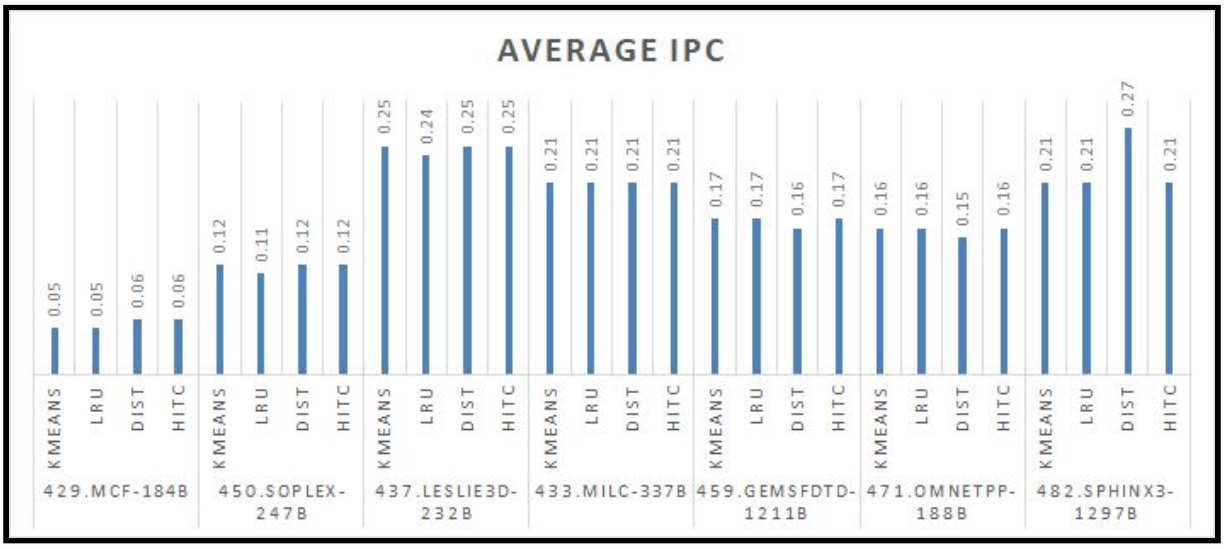
Total Execution Cycle was slightly higher than LRU in three of the seven trace files: Mcf, GemsFDTD, and Omnetpp. However, reduced from Sophlex, Leslie, Milc, and Sphinx3 trace files, increasing efficiency by an average of 5%.



The most advanced algorithm for improving the efficiency of the Total Execution Cycle was the DIST algorithm. For K-means, the performance was also lower than the LRU if sufficient data were not available.

3-2. IPC

Trace File	Algorithm	Average IPC	Trace File	Algorithm	Average IPC
429.mcf-184B	KMEANS	0.05	459.GemsFDTD-1211B	KMEANS	0.17
	LRU	0.05		LRU	0.17
	DIST	0.06		DIST	0.16
	HITC	0.06		HITC	0.17
450.soplex-247B	KMEANS	0.12	471.omnetpp-188B	KMEANS	0.16
	LRU	0.11		LRU	0.16
	DIST	0.12		DIST	0.15
	HITC	0.12		HITC	0.16
437.leslie3d-232B	KMEANS	0.25	482.sphinx3-1297B	KMEANS	0.21
	LRU	0.24		LRU	0.21
	DIST	0.25		DIST	0.27
	HITC	0.25		HITC	0.21
433.milc-337B	KMEANS	0.21			
	LRU	0.21			
	DIST	0.21			
	HITC	0.21			

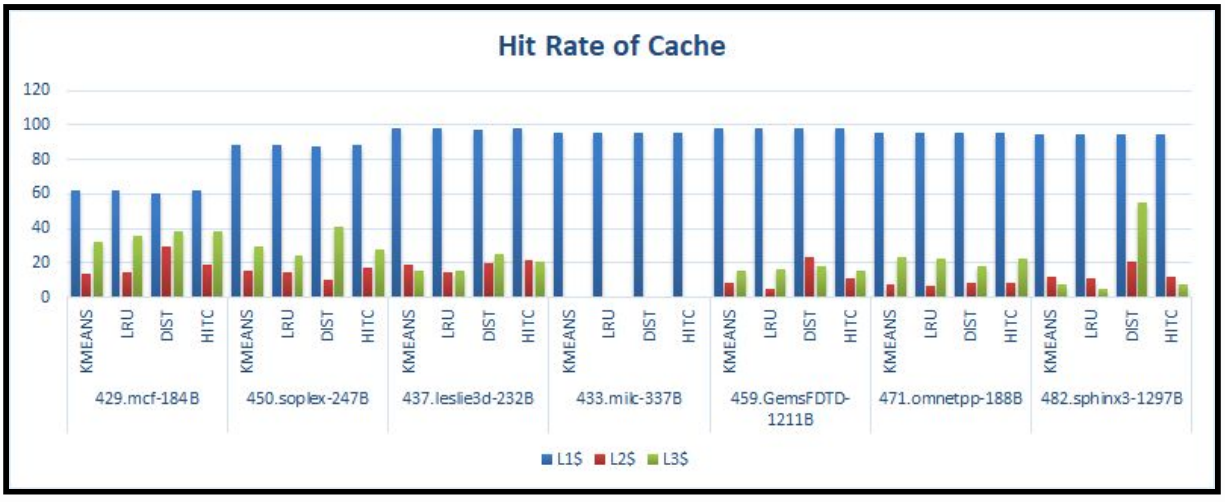


Overall, IPCs show similar or slightly improved performance to LRUs. DIST algorithm recorded 1% lower IPC than LRU algorithm in GemsFDTD and Omnetpp trace file, but improved performance by 6% in Sphinx3 algorithm. Based on the geometric mean, we recorded about 7% improvement in IPC performance compared to LRU algorithm.

3-3. Cache Hit Rates

Trace File	Algorithm	L1$	L2$	L3$	Trace File	Algorithm	L1$	L2$	L3$
429.mcf-184B	KMEANS	61.959	13.702	32.621	459.GemsFDTD-1211B	KMEANS	98.466	8.34	15.234
	LRU	62.355	15.012	35.44		LRU	98.513	5.113	16.387
	DIST	60.283	29.796	38.789		DIST	97.708	23.222	17.896
	HITC	62.002	18.683	38.81		HITC	98.417	11.081	15.879
450.soplex-247B	KMEANS	88.329	15.663	29.885	471.omnetpp-188B	KMEANS	95.361	7.56	23.129
	LRU	88.498	14.738	24.16		LRU	95.405	7.124	22.552
	DIST	87.386	10.012	41.213		DIST	95.17	8.188	18.477
	HITC	88.159	16.991	27.71		HITC	95.339	8.779	22.483
437.leslie3d-232B	KMEANS	97.83	19.24	15.884	482.sphinx3-1297B	KMEANS	94.648	12.125	7.616
	LRU	97.928	15.056	15.613		LRU	94.757	11.613	4.804
	DIST	97.642	19.579	25.078		DIST	94.361	21.292	55.134
	HITC	97.806	21.557	20.884		HITC	94.643	12.171	7.45
433.milc-337B	KMEANS	95.713	0.282	0.008					
	LRU	95.725	0	0.0006					
	DIST	95.722	0.053	0.219					
	HITC	95.716	0.209	0.008					

The results of our research goal, LLC Hit rate, are as above table. In the Sphinx3 trace file, DIST algorithm recorded a hit rate, which is about 51%p higher than the LRU algorithm, and a rise of about 25%p in the Sophlex trace file. In the Omnetpp trace file, both DIST and HITC algorithm compared to the LRU algorithm had slightly decreased Hit rates, but recorded higher Hit rate at the K-means algorithm.

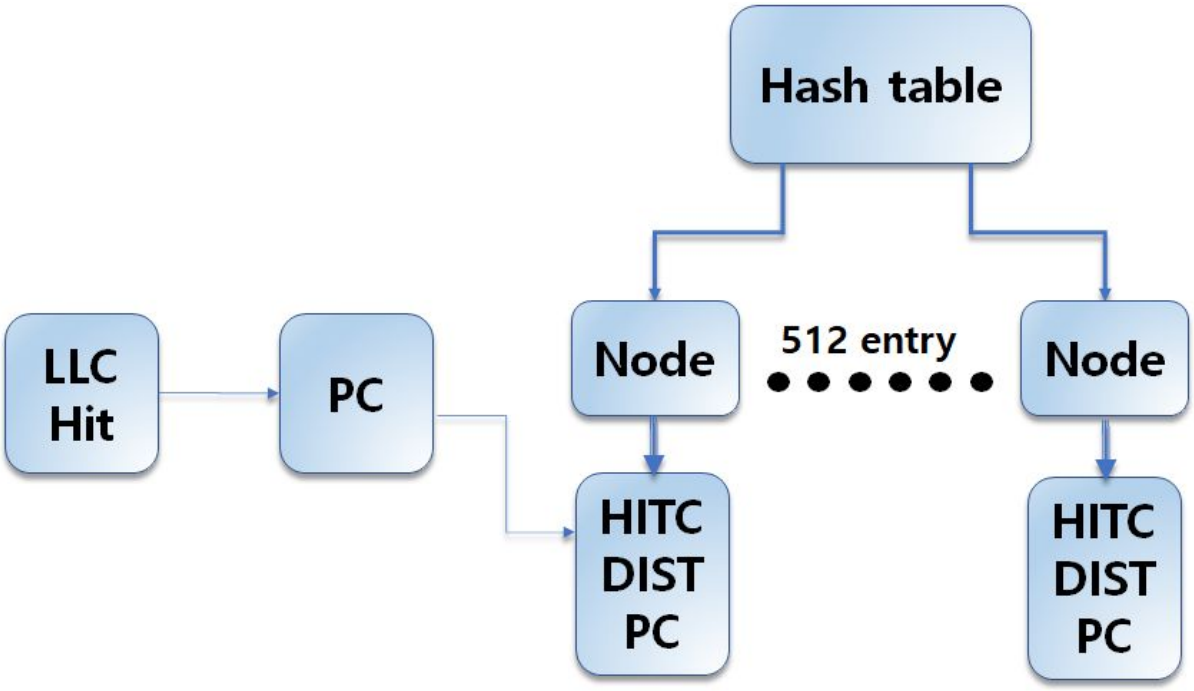


As shown by the graph above, Team Cacher's three algorithms compared to the existing LRU algorithm recorded an average of 6.5% increase in geometric mean, LLC Hit rate. Interestingly, not only LLC, but also L2 cache's hit rate has risen. This is assumed to be due to

the efficient change in the replacement scenario as the Cache replacement policy changes. There is no improvement in performance compared to LRU algorithm in the Milc trace file. This is because the high Hit rate of L1 cache did not allow access to L2 and L3 cache.

4. Conclusion & Future Advances

4-1. Summary



As mentioned above, replacement policy is critical for performance of computer. We tried to implement replacement policy to enhance LLC hit rate with higher IPC compared with LRU. Since existing algorithms with high performance like hawkeye algorithm have large overhead, our aim was to make practical replacement policy with less overhead. We saved pc value when LLC hit occurs in hash table. All of the algorithm we made use this information in the table to know how cache friendly data it is. Hitc algorithm use information how many LLC hit occured to entry with the pc value. If the hit count value is higher than average, rrpv value of the entry is not increased. Reuse distance algorithm saves the cycle when last entry of the pc value has hit in hash table. If reuse interval is lower than average, rrpv value of the entry is not increased. K-means algorithm uses both hit count data and reuse distance data for entry with

the pc value in hashtable. With these replacement policy, we got higher performance than LRU at most of benchmarks while having reasonable overhead.

4-2. Possible Enhancements

We can apply AI logics to logic circuit design. Neural network is possible alternative. We can enter hit count, reusing distance and pc value to get proper rrpv value. It need to be trained by some benchmarks. After training, the replacement policy will have good performance.

5. Reference

[Web]
Cache image from https://www.undocopy.com/2017/07/what-is-cache-memory-types-and.html

[Paper]
High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP) - Aamer Jaleel† Kevin B. Theobald‡ Simon C. Steely Jr.† Joel Emer†

Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement(2016) - Akanksha Jain, Calvin Lin.