CS 232- Lab5 VTUNES CHAMPSIM

Parth Pujari, Anish A. Kulkarni, Ramswroop Rathi

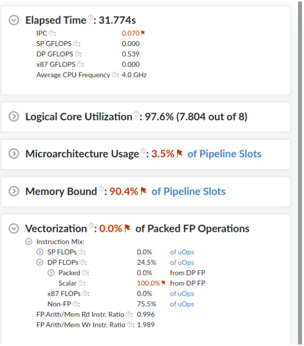
IIT Bombay

April 2023

Table of Contents

Performance Snapshot

- IPC : 0.07
- Bad Speculation : 0.1% of pipelined slots
- Logic Core Utilisation : 7.804 out of 8



Hotspot Detection

- Hotspot Function : `Multiply1`
- CPU Time : 100%

Elapsed Time: 24.638s

CPU Time:	190.010s
Total Thread Count:	9
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typi application performance.

Function	Module	CPU Time	% of CPU Time
multiply1	matrix	190.000s	100.0%
init_arr	matrix	0.010s	0.0%

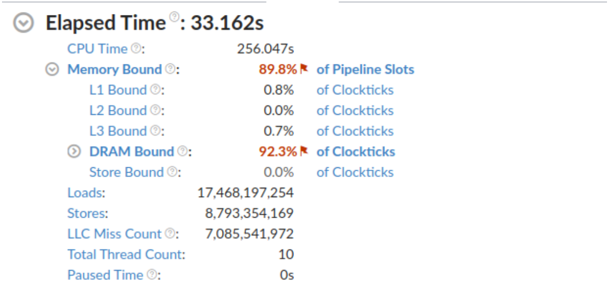
*N/A is applied to non-summable metrics.

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simult

Memory Access Analysis

- L1 Bound : 0.0%
- L2 Bound : 0.7%
- LLC Bound : 92.3%



Elapsed Time: 33.162s

CPU Time:	256.047s	
Memory Bound:	89.8%	of Pipeline Slots
L1 Bound:	0.8%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	0.7%	of Clockticks
DRAM Bound:	92.3%	of Clockticks
Store Bound:	0.0%	of Clockticks
Loads:	17,468,197,254	
Stores:	8,793,354,169	
LLC Miss Count:	7,085,541,972	
Total Thread Count:	10	
Paused Time:	0s	

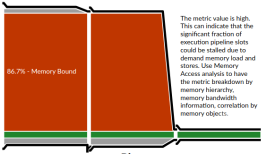
Matrix Multiplication

Microarchitecture exploration

- Instructions retired : 69,729,600,000
- Average CPU Frequency : 4.0GHz
- Effective Logic Core Utilization : 7.780/8

Elapsed Time: 23.607s

Clockticks:	732,919,200,000	
Instructions Retired:	69,729,600,000	
CPI Rate:	10.511	
Retiring:	4.9%	of Pipeline Slots
Front-End Bound:	3.3%	of Pipeline Slots
Bad Speculation:	0.1%	of Pipeline Slots
Branch Mispredict:	0.0%	of Pipeline Slots
Machine Clears:	0.1%	of Pipeline Slots
Back-End Bound:	91.7%	of Pipeline Slots
Memory Bound:	86.7%	of Pipeline Slots
L1 Bound:	0.0%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	10.7%	of Clockticks
Contested Accesses:	0.5%	of Clockticks
Data Sharing:	15.0%	of Clockticks
L3 Latency:	5.7%	of Clockticks
SQ Full:	0.0%	of Clockticks
DRAM Bound:	80.3%	of Clockticks
Memory Bandwidth:	76.1%	of Clockticks
Memory Latency:	22.0%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	5.0%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	4.4%	of Clockticks
Cycles of 0 Ports Utilized:	86.3%	of Clockticks
Cycles of 1 Port Utilized:	7.1%	of Clockticks
Cycles of 2 Ports Utilized:	3.7%	of Clockticks
Cycles of 3+ Ports Utilized:	2.8%	of Clockticks
Vector Capacity Usage (FPU):	25.0%	
Average CPU Frequency:	4.0 GHz	
Total Thread Count:	10	
Paused Time:	0s	



µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Cracker Phase 0

The original performance is as follows

= = = VTune profiling demo = = =

==============================

memory malloc +OK

initializing from built-in data

memory malloc +OK

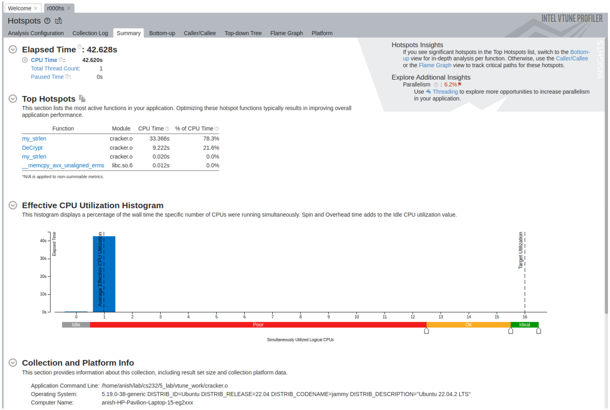
max iter 10000

===

try search... wait!

Password per sec: 210

Cracker Phase 0



Cracker Phase 1

- In the hotspots we saw that my_strlen takes most amount of time
- Hence we first tried to optimize my_strlen
- We are accessing memory byte by byte to find null character
- We can speed up the process by fetching 8 bytes at a time from memory using unsigned long long int

Cracker Phase 1

```c
struct key_attr
{
    int zi;
    long ki;
    char a[2];
    int li;
};
struct key_attr key[KEY_SIZE];

char correct_plaintext[MAX_CRYPT_LEN]; // zeroed out plaintext to do a final check against;

int my_strlen(char *str)
{
    unsigned long long int *x = (unsigned long long int*) str;
    int t=0;
    while(1){

      if((*x&255)==0){
        return t;
      }
      else if((*x&65280)==0){
        return t+1;
      }
      else if((*x&16711680)==0){
        return t+2;
      }
      else if((*x&4278190080)==0){
        return t+3;
      }
      x = x+1;
      t = t+4;
    }
}

// Decrypting ciphertext using the found password
char *DeCrypt(char *ciphertext, char *pswd)
{
    int sub_func(int num)
    {
        switch (num)
        {
        case 0:
            return 1;
```

Cracker Phase 1

= = = VTune profiling demo = = =

====================================

memory malloc +OK

initializing from built-in data

memory malloc +OK

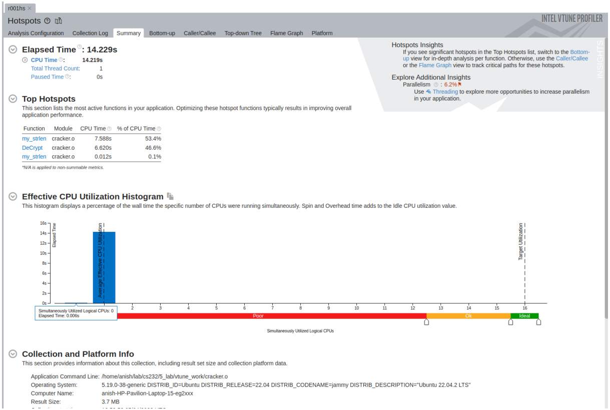
max iter 10000

===

try search... wait!

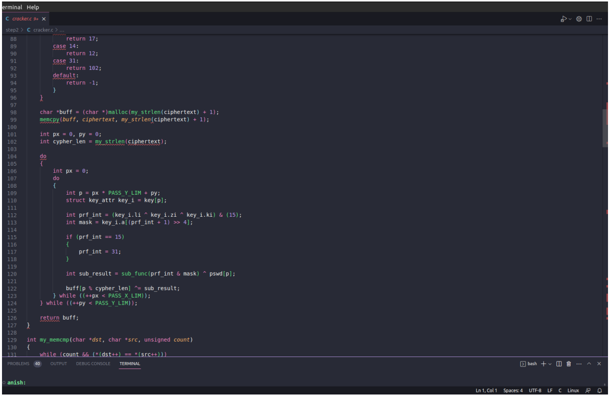
Password per sec: 661

Cracker Phase 1



Cracker Phase 2

- We saw a three times increase after optimising my_strlen
- But still it is the major hotspot, so we check all places where it is called
- We discovered that in Decrypt it was calling my_strlen inside a loop with same argument
- We optimised this by calling it just once outside the loop and storing it as variable



```
            return 17;
        case 14:
            return 12;
        case 31:
            return 102;
        default:
            return -1;
        }
    }

    char *buff = (char *)malloc(my_strlen(ciphertext) + 1);
    memcpy(buff, ciphertext, my_strlen(ciphertext) + 1);

    int px = 0, py = 0;
    int cypher_len = my_strlen(ciphertext);

    do
    {
        int px = 0;
        do
        {
            int p = px * PASS_Y_LIM + py;
            struct key_attr key_i = key[p];

            int prf_int = (key_i.li ^ key_i.zi ^ key_i.ki) & (15);
            int mask = key_i.a[(prf_int + 1) >> 4];

            if (prf_int == 15)
            {
                prf_int = 31;
            }

            int sub_result = sub_func(prf_int & mask) ^ pswd[p];

            buff[p % cypher_len] ^= sub_result;
        } while ((++px < PASS_X_LIM));
    } while ((++py < PASS_Y_LIM));

    return buff;
}

int my_memcmp(char *dst, char *src, unsigned count)
{
    while (count && (*(dst++) == *(src++)))
```

Cracker Phase 2

= = = VTune profiling demo = = =

==================================

memory malloc +OK

initializing from built-in data

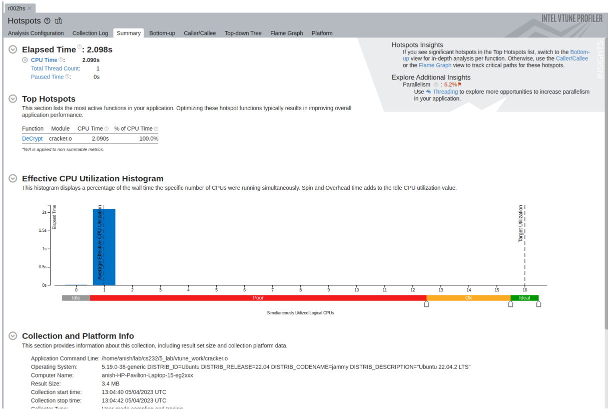
memory malloc +OK

max iter 10000

===

try search... wait!

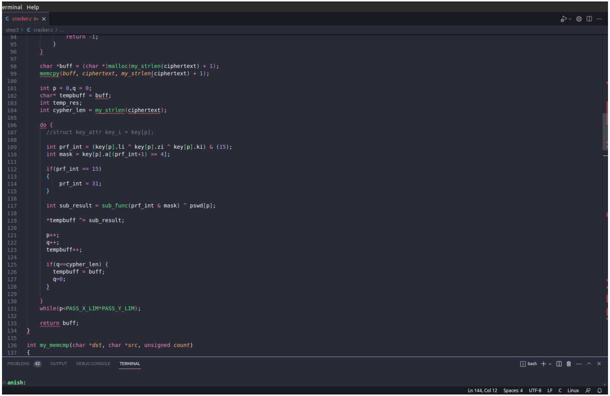
Password per sec: 51,91


r002hs
Hotspots
Analysis Configuration Collection Log Summary Bottom-up Caller/Callee Top-down Tree Flame Graph Platform
INTEL VTUNE PROFILER
Elapsed Time: 2.098s
CPU Time: 2.090s
Total Thread Count: 1
Paused Time: 0s
Top Hotspots
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.
Function Module CPU Time % of CPU Time
DeCrypt cracker.o 2.090s 100.0%
*N/A is applied to non-summable metrics.
Hotspots Insights
If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee or the Flame Graph view to track critical paths for these hotspots.
Explore Additional Insights
Parallelism: 6.2%
Use Threading to explore more opportunities to increase parallelism in your application.
Effective CPU Utilization Histogram
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.
Elapsed Time
Average Effective CPU Utilization
Target Utilization
Idle
Poor
Ok
Ideal
Simultaneously Utilized Logical CPUs
Collection and Platform Info
This section provides information about this collection, including result set size and collection platform data.
Application Command Line: /home/anish/lab/cs232/5_lab/vtune_work/cracker.o
Operating System: 5.19.0-38-generic DISTRIB_ID=Ubuntu DISTRIB_RELEASE=22.04 DISTRIB_CODENAME=jammy DISTRIB_DESCRIPTION="Ubuntu 22.04.2 LTS"
Computer Name: anish-HP-Pavilion-Laptop-15-eg2xxx
Result Size: 3.4 MB
Collection start time: 13:04:40 05/04/2023 UTC
Collection stop time: 13:04:42 05/04/2023 UTC

Cracker Phase 3

- Now the hotspot is in the Decrypt function.
- The main contributing factors seems to be towards the end of the loop
- So we make two modifications, turn the two loops into a single loop and instead of taking % we simply keep track of the proper index at all times.
- Also at the start of the loop key[p] is copied into a fresh structure key_attr, this takes unnecessary time by calling copy constructor
- So instead we just access key[p] directly

Cracker Phase 3



```c
                return -1;
            }
        }

        char *buff = (char *)malloc(my_strlen(ciphertext) + 1);
        memcpy(buff, ciphertext, my_strlen(ciphertext) + 1);

        int p = 0,q = 0;
        char* tempbuff = buff;
        int temp_res;
        int cypher_len = my_strlen(ciphertext);

        do {
          //struct key_attr key_i = key[p];

          int prf_int = (key[p].li ^ key[p].zi ^ key[p].ki) & (15);
          int mask = key[p].a[(prf_int+1) >> 4];

          if(prf_int == 15)
          {
              prf_int = 31;
          }

          int sub_result = sub_func(prf_int & mask) ^ pswd[p];

          *tempbuff ^= sub_result;

          p++;
          q++;
          tempbuff++;

          if(q==cypher_len) {
            tempbuff = buff;
            q=0;
          }

        }
        while(p<PASS_X_LIM*PASS_Y_LIM);

        return buff;
    }

    int my_memcmp(char *dst, char *src, unsigned count)
    {
```

Cracker Phase 3

= = = VTune profiling demo = = =

====================================

memory malloc +OK

initializing from built-in data

memory malloc +OK

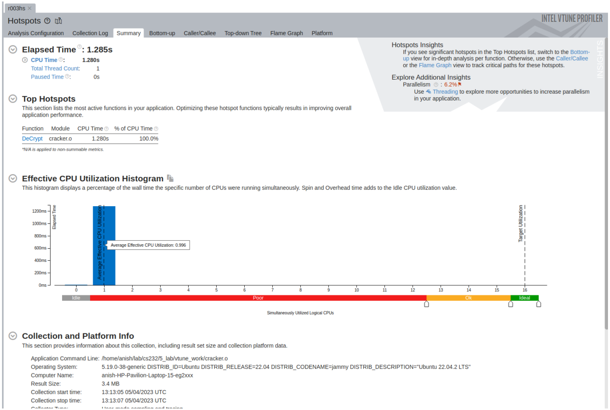
max iter 10000

===

try search... wait!

Password per sec: 94,17

Cracker Phase 3



Question 1

The `handle_fill()` function

- This function is called when a new fill comes from a lower level cache in response to a request sent by the current cache.
- First, a victim is found in the corresponding set to be removed.
- If this victim line is dirty, i.e., it has been written to directly in the cache, a write-back request is sent.
- Writeback request is necessary because the line currently stores updated data which is not present in the lower caches.
- Note that if there is not enough space for the write-back request in the lower level cache queue we need to stall.
- If the victim is not dirty it is replaced by the mew fill

Question 1

`Fill Level`

- Fill_level of a cache is a level associated with each cache, and this field in the packets stores which cache to fetch the data to.

Cache Level	L1	L2	LLC	DRAM
Fill_level	1	2	4	16

- If the fill level of the packet is less than the fill level of the current cache then the data is forwarded to the next higher level cache.

Question 1

`Set and Way`

- Finding a set from address is really easy, in cache.cc there is a function get_set for the same.
- In `get_set`, we take last log2(NUM_SET) bits of address and get the corresponding set.
- For way, if it is a hit then way is the corresponding location of the hit.
- If it is a miss, request is sent to lower cache for the data.
- After this data arrives we call replacement policy function for the respective cache to decide which victim to eject and insert the received data.

Question 1

`return_data`

- The return_data returns data from a lower cache to a higher cache in response to a data request
- First `check_mshr` function is called to check if MSHR still contains the request.
- Then the data is stored in MSHR.data(for the corresponding index) and MSHR.returned is set to COMPLETED. This data is later copied into the cache in cache_fill.
- Next, update_fill_cycle is called to find the next index from MSHR to be called.

Cache Replacement Policies

Least Frequently Used

Implemented by keeping track of the number of times an address got a hit.

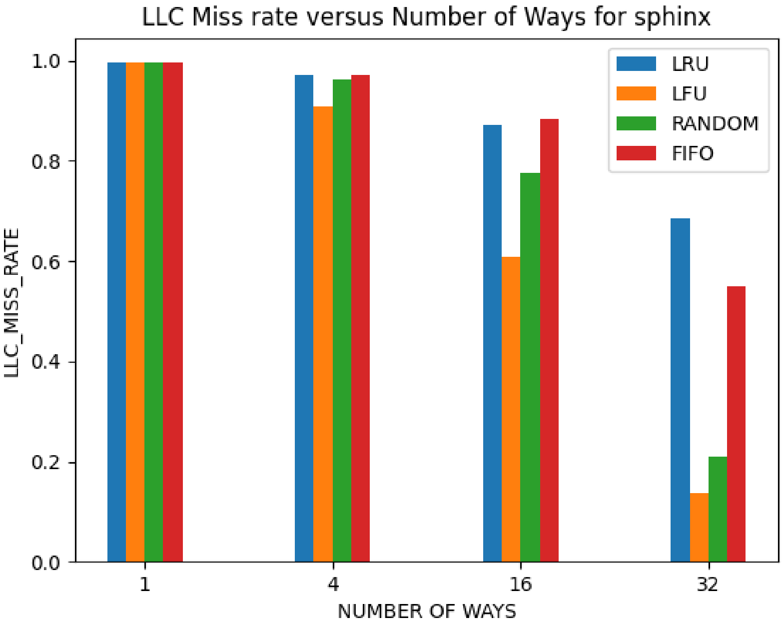
Cache Replacement Policies

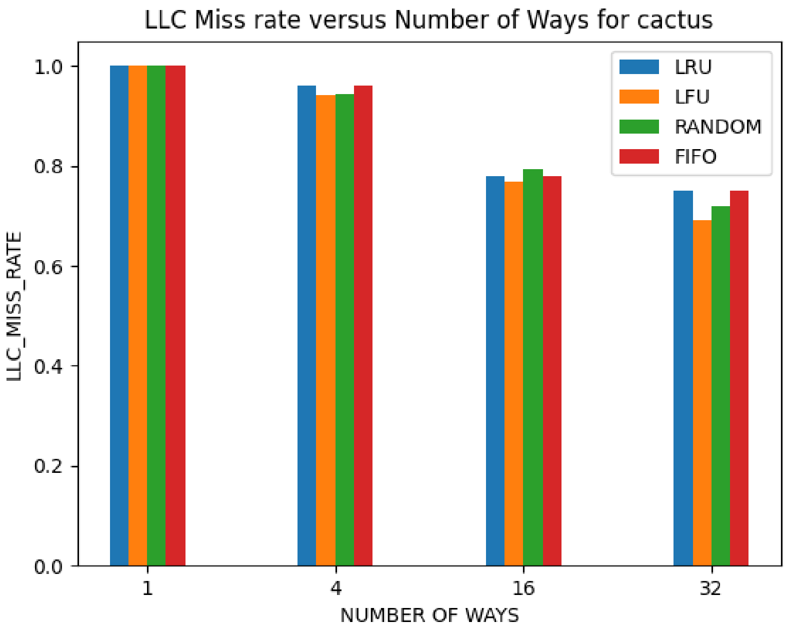
First in First Out
Implemented by keeping a queue and removing the end of the queue which was inserted first.

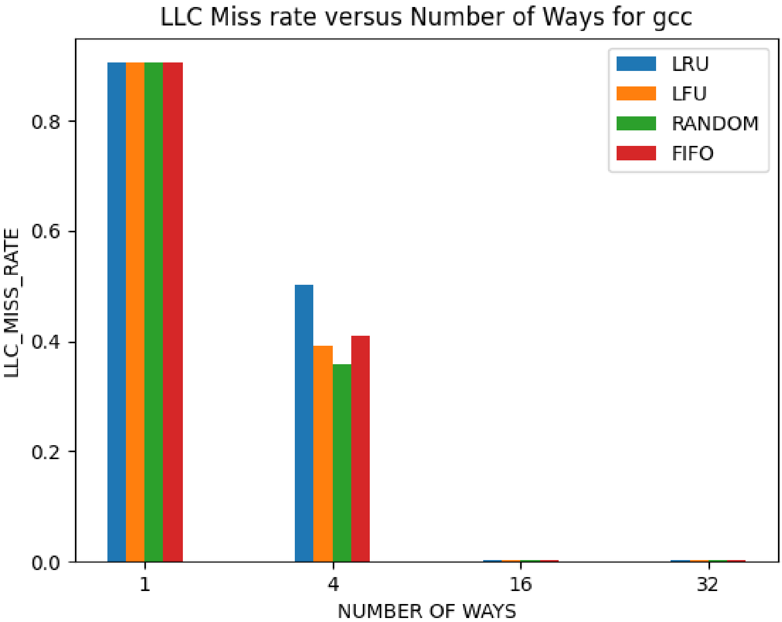
Cache Replacement Policies

Random Replacement

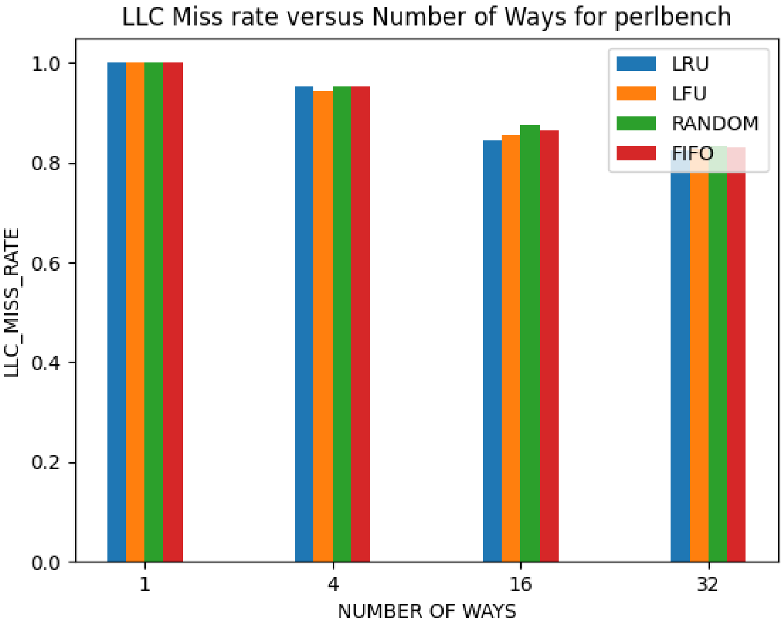
Implemented by choosing a random way in the given set from a uniform distribution in case of a miss.

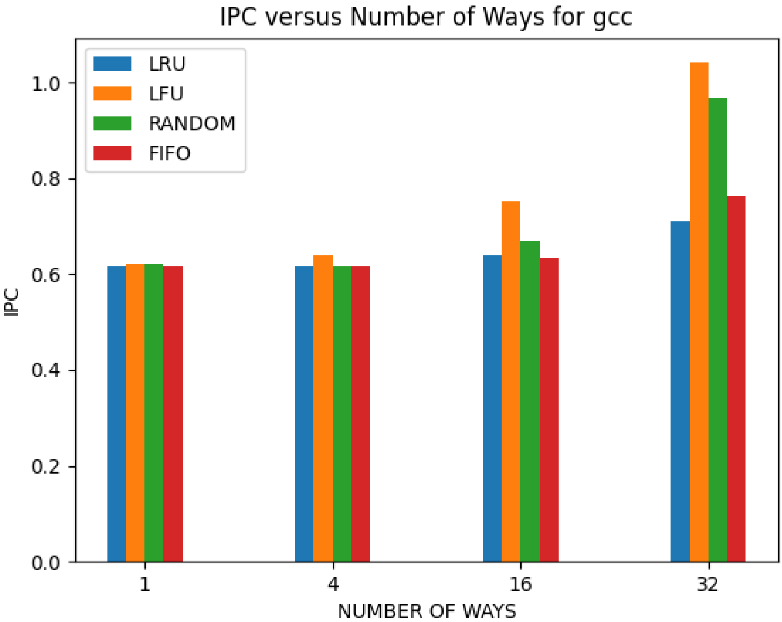

LLC Miss rate versus Number of Ways for sphinx
LLC_MISS_RATE
0.0
0.2
0.4
0.6
0.8
1.0
1
4
16
32
NUMBER OF WAYS
LRU
LFU
RANDOM
FIFO


LLC Miss rate versus Number of Ways for cactus
LLC_MISS_RATE
0.0
0.2
0.4
0.6
0.8
1.0
1
4
16
32
NUMBER OF WAYS
LRU
LFU
RANDOM
FIFO



LLC Miss rate versus Number of Ways for gcc
LRU
LFU
RANDOM
FIFO
LLC_MISS_RATE
0.0
0.2
0.4
0.6
0.8
1
4
16
32
NUMBER OF WAYS


LLC Miss rate versus Number of Ways for perlbench
LRU
LFU
RANDOM
FIFO
1.0
0.8
0.6
0.4
0.2
0.0
LLC_MISS_RATE
1
4
16
32
NUMBER OF WAYS


IPC versus Number of Ways for gcc
LRU
LFU
RANDOM
FIFO
1.0
0.8
0.6
0.4
0.2
0.0
IPC
1
4
16
32
NUMBER OF WAYS

Inferences

- We see that LFU is clearly the best policy. Random is at second place followed by a LRU and FIFO.
- LFU's win can be explained by the fact that it has good memory. For example if a certain address was used a lot of times in past and hasn't been used in a while LFU will remember it but other policies will not.
- FIFO and LRU are bad because they may kick out memory addresses used with high frequency for more recent low frequency addresses.
- Random is in middle because it doesn't have a preference and the high frequency addresses have a chance of staying

Fixed Offset Prefetchers

- Generalization of next line prefetchers
- Prefetch X+1 in case of an L2 miss for X or has been prefetched into L2 but not fetched by L1
- Offset prefetchers fetch a line at a fixed offset from X

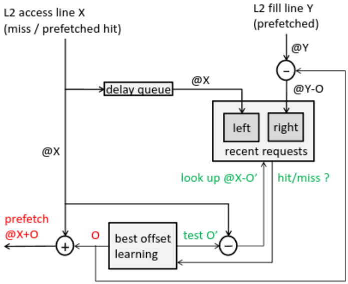
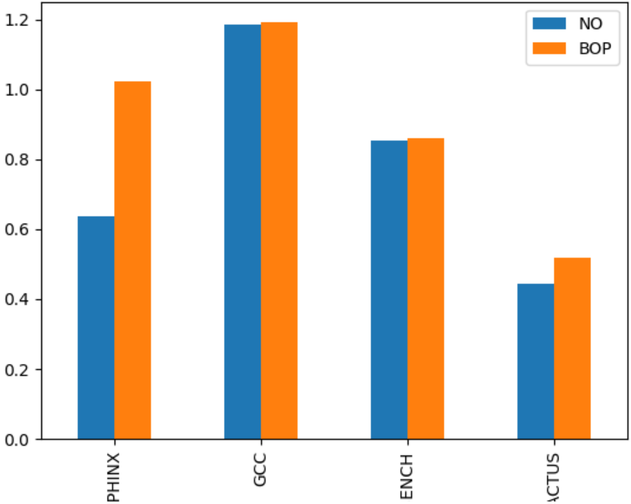


Figure 1: **The BO prefetcher**

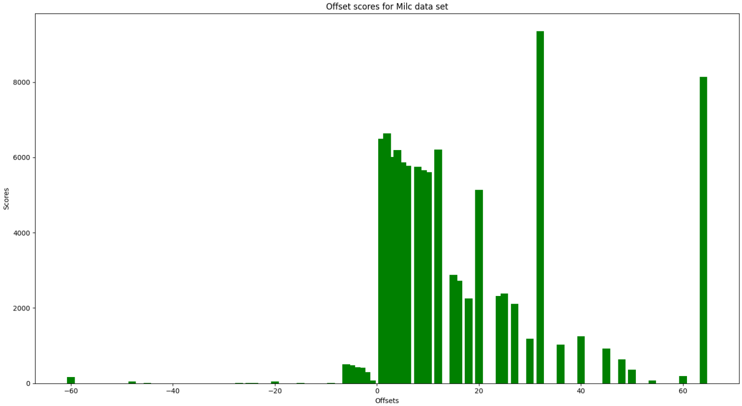
Best Offset Prefetchers

- The aim is to find the best possible offset to prefetch from
- Keep a current offset O
- Keep a recent requests table
- Store a list of offsets (O_i) to choose from, each with a score
- The following happens in what is called a *learning phase*
- A round begins, set i to 0
- For every L2 miss or prefetched hit X, we check if X- O_i is in the RR table
- If so, increase the score of O_i, else don't and store X after a delay
- If Y is prefetched, store Y-O in RR table
- If score reaches Score MAX or rounds reach Round MAX at the end of a round, a phase ends

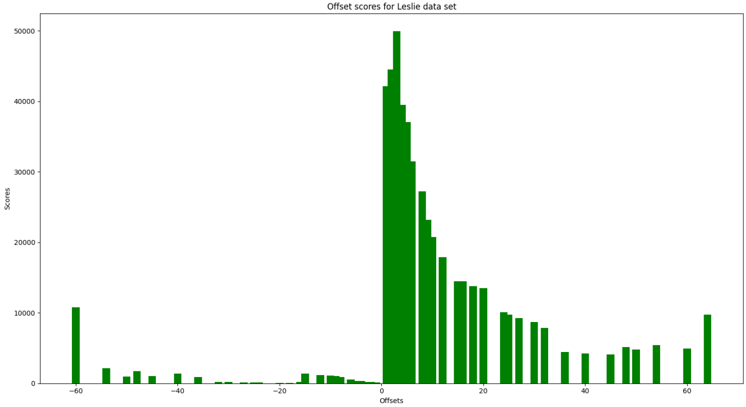
Best Offset Prefetchers



Best Offset Prefetchers



Best Offset Prefetchers



Best Offset Prefetchers: Conclusion and Optimizations

We concluded the following:

- The optimal value for SCORE_MAX is 40 and that of ROUND_MAX is 50.
- We tried searching for all X-O_is (for all 'i's in one round instead of just 1). However, this resulted in a decrease in IPC.