Performance measurement of Intel Corei5 by Recurrent Neural Network

Muhammad Hasnain Khan
School Of Computing
FAST NUCES Islamabad
i191255@nu.edu.pk

***Abstract*—Natural Language Processing a way used in computers to understand the human language. In deep learning, one of the most commonly used neural network for generating some meaningful text is RNN (Recurrent Neural Network) that is developed and maintained by Google Brain Team. RNN (Recurrent Neural Network) is a type of neural network where the output of previous hidden layers are given to the next hidden layers in order to generate the correct paragraph from the words that were learned from the input dataset. Automatic summarization, topic segmentation, translation and sentiment analysis can be done using RNN. In this paper, we will be evaluating the performance of RNN in terms of memory utilization for Intel Corei5 processor with and without GPU.Different experiments are performed to evaluate RNN with different combinations of hardware i.e with and without L2 cache, GPU.**

***Index Terms*—Cache, Optimization, mapping technique, RNN**

I. Introduction

The fastest memory of our computers is Cache but the size is less. And because of the size limitations we have to move and delete blocks and sets from cache according to the needs. We can't do processing on the data that is placed in the main memory because that will be a lot costly in terms of times so we have to have a mechanism of taking blocks from main memory to cache. It can be seen from the image that can be found below that the transfer of main memory to cache is super slow.

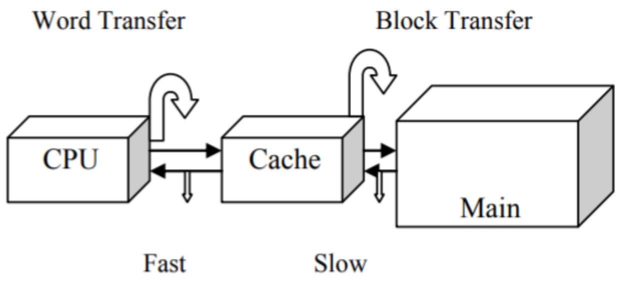


Fig. 1. Working of Cache.

Program behavior analysis is one of the most important methods for computer architecture design, system software optimization and application performance improvements. As the gap between processor and memory is increasing day by day that's why it has been a hot issue to eliminate the **memory wall** problem for long. Thus, memory behavior/trend is especially significant for analyzing program execution behaviors. Simulation and profiling are the most common and efficient approaches to obtain memory behavior. In this paper, different profiling tools and simulators are used to obtain the memory behavior like memory traces.

Experiments were performed in the below order; changing in Cache size along replacement policy and set size. Cache replacement policy is the most important design parameter and the processor performance can be affected. The principal of locality of reference is used to get data or instructions of program. In this specific situation, analysts today have progressively moved their attention on the streamlining of these ML calculations. A significant concern depends on the reason that for the creation of productive outcomes, ideal abuse of the equipment is required. That equipment incorporates and isn't constrained to: CPU, Cache, Main Memory, and Secondary storage. Continuous frameworks are significantly increasingly subordinate upon the productive usage of these assets as they have both constrained assets and time-basic nature that intensifies the multifaceted nature of the issue. This compares to the significance of this exploration territory as any client today would like to utilize not just the effective code usage of the ML calculation yet additionally the way wherein that code misuses equipment assets of the framework.

It is apparent that to accomplish this degree of advancement where a conventional code might be intended to ideally abuse equipment of a framework is a multifaceted assignment. There are various reasons why this is the situation; first, programming dialects have their own arrangements of assemblage decides and guidelines that do enable different frameworks to accomplish similarity however not streamlining. Second, the developers are not constantly mindful of the framework equipment setups on which their code should run, for instance, code is composed to store and recover information from memory in push significant request while the equipment on which it is run may incline toward segment significant request. Third, the regularly changing and dynamic nature of the processor engineering additionally has a critical bearing on the general condition.

A significant part of the work that is molding research in this field rotates around execution assessment on different CPUs and GPUs. SPEC benchmarks are likewise used to perform broad relative examination of different frameworks. In this paper, we have guided our push to run open source python execution of two of the generally utilized ML calculations: Decision Tree and RNN. Different instruments of CPU investigation, memory age examples and reserve reenactment are utilized to produce factual outcomes. Moreover, reference diagrams are also utilized to extend the results of these tests.

II. Related Work

Neural networks have emerged as a powerful technique to address sequence prediction problems and Neural Networks has has been a really bare bone of every deep learning and convolutional algorithm and it can used to solve natural language processing (NLP) and text understanding problems. The Recurrent Neural Network (RNN) is neural sequence model that achieves state of the art performance on important tasks that include language modeling [1], speech recognition [2], and machine translation [3]. Neural networks are being used in every deep learning problem and because of this, the imposed trend state a significant challenge of dropping the prediction accuracy too deep, when the working set is larger than the predictive table. Scaling predictive tables with fast-growing working sets is difficult and costly for hardware implementation that's why in this paper, we are going to evaluate the RNN with different combination of hardware spec. Prefetching is a regression problem itself. However the outer space is extremely sparse and vast and it is a poor fit for standard regresiion model. That's why an underlying model could utilize two features at a given timestep N. It could utilize the location and PC that produced a cache miss at that timestep to anticipate the location of the miss at timestep N + 1. LSTM(Long Short-Term Memory) was composed by a hidden layer h and a cell state c, with input i, forget f, and output gates o that dictate what information gets stored and propagated to the next timestep [4]. LSTM is being used by different variants of RNN that is designed to learn long, complex patterns within a sequence. If a pattern have non-linear correlation having elements in sequence then that sequence will be called complex sequence. We used LSTM with RNN because till now LSTM is the only one which shows the good and promising results related to our problem which is cache utilization.

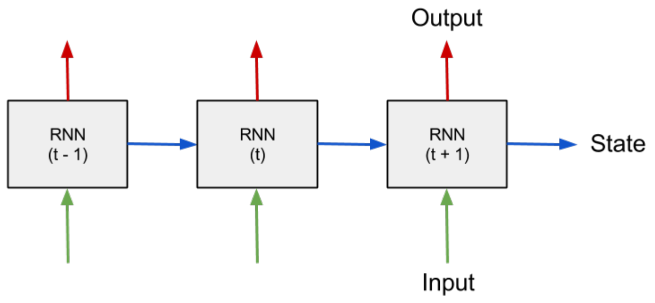


Fig. 2. RNN in TensorFlow

A new **Neural Cache** can also be implemented Using LSTM. To achieve the best performance out of RNN with LSTM, there are a number of architectural variants OF RNN that gives high confidence score of better performance on the problems that uses long term dependencies. In this paper, four caches were used to speed up RNN [5]. Cache had been implemented using hash tables and these hash tables were used to store key along with pairs. Report cache can also be used for prefetching as it just stores the information that was recently requested from the source. Whenever a store is requested for the first time then a new report cache will be created and after that significant improvement in results can be seen with report cache as prefetching is enabled. But creating a new report cache will be costly. A study shows a simple modification in RNN to the standard RNN architecture (clockwork RNN), in which the hidden layer is partitioned into separate module, each processing input at its own temporal singularity and making computations only at its prescribed clock rate. As opposed to making the standard RNN models progressively mind boggling, CW-RNN decreases the number of RNN parameters, improves the exhibition altogether in the undertakings tried, and accelerates the arrange assessment. The system is illustrated in starter tests including two assignments: sound sign age and TIMIT verbally expressed word arrangement, where it outflanks both RNN what's more, LSTM systems. There are different vaiants of RNN which different training set data and different algorithms and some of the algorithms are explained below.

A. *KNN*

The K-Nearest Neighbors calculation utilizes the whole informational index as the preparation set, as opposed to parting the informational index into a preparing set and test set. At the point when a result is required for a new information occurrence, the KNN calculation experiences the whole informational index to discover the k-closest cases to the new occurrence, or on the other hand, the k number of cases generally like the new record, and afterward yields the mean of the results or the mode (most successive class) for an order issue. The estimation of k is the client indicated.

B. *Wordnet PCA*

Principle Component Analysis (PCA) is utilized to make information simple to investigate and picture by decreasing the quantity of factors. This is finished by catching the most extreme difference in the information into another organize framework with axes called **principle segment**. Every part is a direct mix of the first factors and is symmetrical to one another. The symmetry between parts demonstrates that the the connection between these parts is zero. Each word is connected to a vector and then each vector is connected to a layer which holds the words related to that vector word.

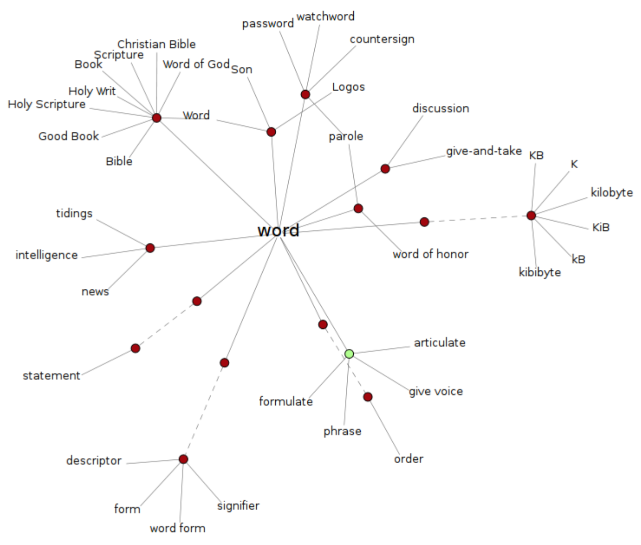


Fig. 3. Wordnet PCA

C. K-means

K-means is an iterative calculation that gatherings comparative information into clusters. It figures the centroids of k bunches and doles out an information point to that bunch having a minimal separation between its centroid and the information point. We start by picking an estimation of k. Here we will take assumption of k 5. At that point, we arbitrarily allocate every datum point to any of the 3 bunches. Process group centroid for every one of the bunches. The red, blue and green stars signify the centroids for every one of the 5 groups. Next, reassign each point to the nearest group centroid. In the figure over, the upper centroid indicates got doled out the group with the blue centroid. Pursue a similar strategy to allot focuses on the bunches containing the red and green centroids. At that point, compute centroids for the new bunches. The old centroids are dark stars; the new centroids are red, green, and blue stars. At long last, rehash stages 4-5 until there is no exchanging of focuses from one group to another. Once there is no exchanging for 2 successive advances, leave the K-implies calculation

Each cluster is associated with one centroid.

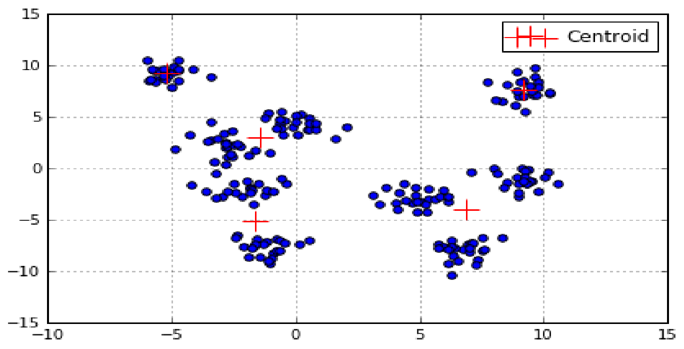


Fig. 4. K-means

III. Experimental Details

A. Experiment Setup

In the following section, experimental details are provided of an open source deep learning neural network RNN. Different tools, hardware combinations and simulators are used for testing the performance of RNN.

B. Hardware Platform

The tables given below shows the system specs along memory information.

Processor Name	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
Model	142
Architecture	x86_64

Fig. 5. System Specs

Cache	Cache Type	Number Of Sets	Size	Ways of associativity
Cache L1d	Data Cache	63	32KB	7
Cache L1i	Instruction Cache	63	32KB	7
Cache L2	Unified Cache	1023	256KB	3
Cache L3	Unified Cache	4095	3072KB	6
RAM	DDR4	None	8192MB	None

Fig. 6. System's Memory Specs

C. Data Set

The Sun Also Rises novel has been used as a input to the model. The neural network has been trained on this novel and then that trained model has been used to generate an paragraph having zero fault tolerance i.e grammar mistake and the paragraph has to be contextually correct.

D. Tools

Various tools have been used to comprehend the working of the RNN on the x86_64 architecture. Perf tool has been used to get to know the total event count, after that perf stat is used to collect multiple stats such as i) Instruction per cycle (IPC) ii) CPU utilization iii) Stalled Cycles per Instruction iv) Context Switches v) Page Faults vi) Branch misses vii) Branch hits etc. To understand the memory generation text, I used **Valgrind**. Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. . These memory addresses are stored in a file, separately, for both python codes and **Massif Visualizer** is used to generate heap profile of both of these codes. Although the exact massif representation is questionable as the memory trace files were too small to give an overall view of the working of the program, yet for learning purposes it did prove useful.

Furthermore, I also used **cProfile**, a python profiling tool, to see the number of function calls made by the code during the time of their execution. In conjunction with cProfile, we also use **Python Call Graph** to delve deeper into the relationship between various subroutines in ML programs under consideration.

Aside from the above tools, **Cache Simulator** has been used for understanding the importance of varying the cache sizes, replacement policies and by enabling and disabling prefetching. We performed 21 sets of experiments by using different configurations of hardware. Some experiments are performed with and without L2 cache and the impact of disabling L2 cache was also analyzed thoroughly. A different kind of results were achieved by disabling prefetch. Results were better if prefetching is disabled. After studying about cache and RNN performance, we came to know that this may be more because of the shared L2 cache than the shared L1 instruction cache, where we wouldn't expect much if any writing. The L1 data cache, where we expect plenty of writing, is not shared.

Different experiments were performed to comprehend the conduct of memory for RNN calculation. As a matter of first importance CSV document has been produced by executing RNN calculation code on the machine. Calculations has been recorded by running RNN code for some time and a CSV parallel record has been created. In CSV document various Data set have been executed after usage of memory. As various quantities of load, store, instructions and memory utilization at various memory address spaces with various bytes size have been produced.
Than valgrind tool have been used for memory traces and after that a bar chart of memory traces i.e load, store, instructions and write back has been generated by using the data of memory traces which was stored in CSV.

The CSV file has various kinds of information like load, stores, instructions and write back. Above all else separate each a singular informational index like instructions are isolated structure load, store and memory and afterward further recognize instructions on the premise of size than this procedure has been rehashed with load, store and memory. All the data has been isolated and adjusted by using the **Sublime** tool. **MS Excel** has been used to draw the bar chart from the CSV file. When CSV document esteems embedded into exceed expectations after arrangement above chart know as memory visual chart has been drawn. After performing these tasks on RNN, a cache simulator has been used to get the information of the cache utilization. Cache simulator code has been changed according to the different combinations and modifications. After altering the code as by changing policies, bit size, block set number, block size, with fetch, without fetch different memory total access count and average memory access time have been obtained which actually determining the behavior of the memory.

A memory heap consumer visualizer(Massif) tool has been used to generate the graph and after that the graph was analyzed. Graph can be seen below. Massif visualizer can be used for the detection of memory leaks, finding expensive leaks and finding locations which contributes significantly to the memory of the memory consumption of the application.

Bar chart can be seen below.

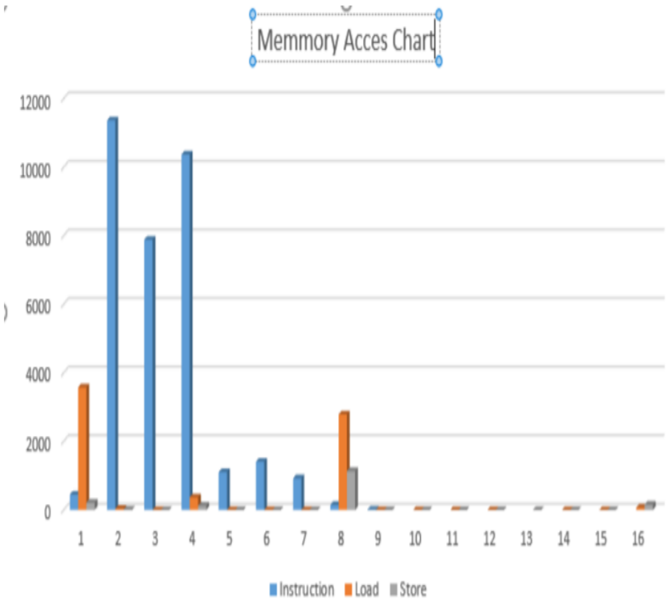


Fig. 7. Memory Access Graph

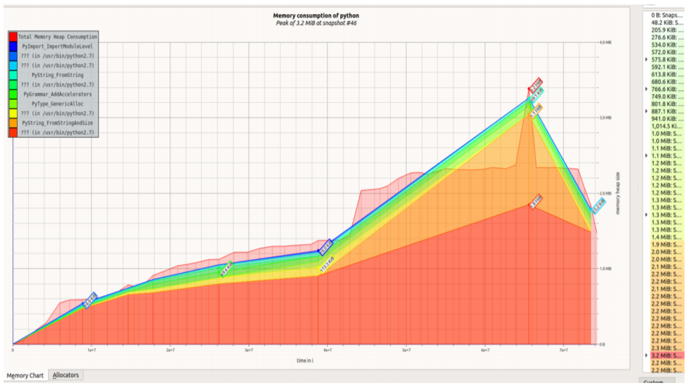
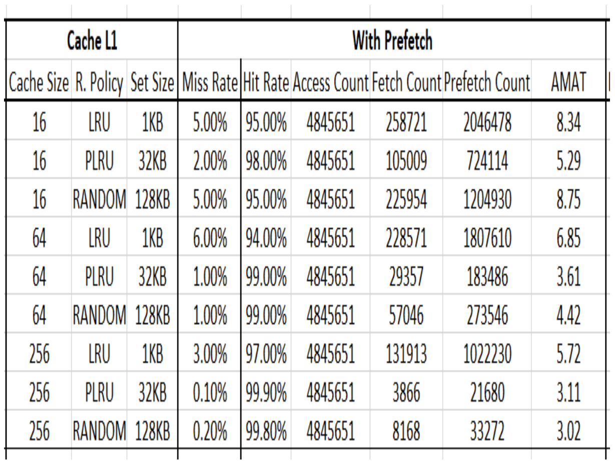


Fig. 8. Heap Consumption Graph

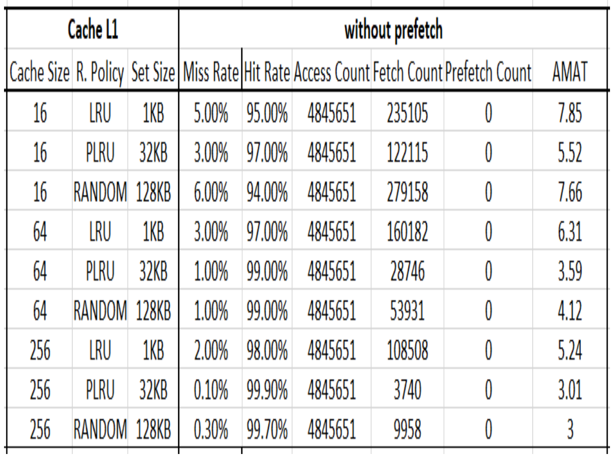
We performed 10 experiments without L2 cache with and without prefect. Hit rate, miss rate, access count, fetch count, prefetch count and average memory access time has been noted by using the cache simulator. Experiments configurations without L2 and with prefetch and results can be seen in the below image.



Cache L1			With Prefetch					
Cache Size	R. Policy	Set Size	Miss Rate	Hit Rate	Access Count	Fetch Count	Prefetch Count	AMAT
16	LRU	1KB	5.00%	95.00%	4845651	258721	2046478	8.34
16	PLRU	32KB	2.00%	98.00%	4845651	105009	724114	5.29
16	RANDOM	128KB	5.00%	95.00%	4845651	225954	1204930	8.75
64	LRU	1KB	6.00%	94.00%	4845651	228571	1807610	6.85
64	PLRU	32KB	1.00%	99.00%	4845651	29357	183486	3.61
64	RANDOM	128KB	1.00%	99.00%	4845651	57046	273546	4.42
256	LRU	1KB	3.00%	97.00%	4845651	131913	1022230	5.72
256	PLRU	32KB	0.10%	99.90%	4845651	3866	21680	3.11
256	RANDOM	128KB	0.20%	99.80%	4845651	8168	33272	3.02

Fig. 9. Without L2

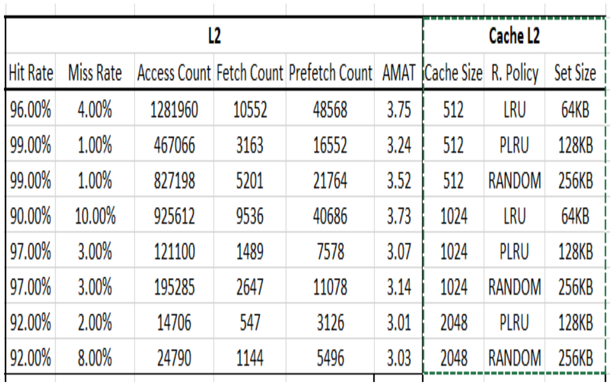
Experiments configurations without L2 and without prefetch and results can be seen in the below image.



Cache L1			without prefetch					
Cache Size	R. Policy	Set Size	Miss Rate	Hit Rate	Access Count	Fetch Count	Prefetch Count	AMAT
16	LRU	1KB	5.00%	95.00%	4845651	235105	0	7.85
16	PLRU	32KB	3.00%	97.00%	4845651	122115	0	5.52
16	RANDOM	128KB	6.00%	94.00%	4845651	279158	0	7.66
64	LRU	1KB	3.00%	97.00%	4845651	160182	0	6.31
64	PLRU	32KB	1.00%	99.00%	4845651	28746	0	3.59
64	RANDOM	128KB	1.00%	99.00%	4845651	53931	0	4.12
256	LRU	1KB	2.00%	98.00%	4845651	108508	0	5.24
256	PLRU	32KB	0.10%	99.90%	4845651	3740	0	3.01
256	RANDOM	128KB	0.30%	99.70%	4845651	9958	0	3

Fig. 10. Without L2

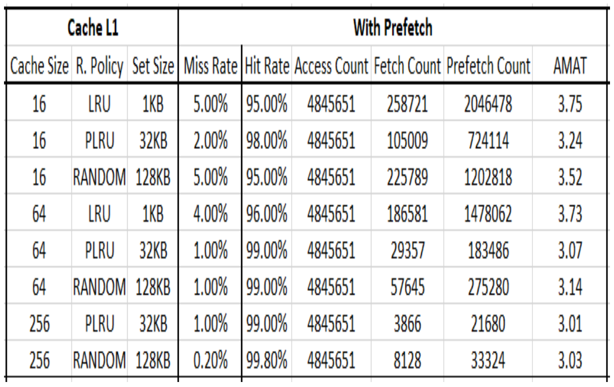
Experiments configurations with L2 and with prefetch and results can be seen in the below image.



L2						Cache L2		
Hit Rate	Miss Rate	Access Count	Fetch Count	Prefetch Count	AMAT	Cache Size	R. Policy	Set Size
96.00%	4.00%	1281960	10552	48568	3.75	512	LRU	64KB
99.00%	1.00%	467066	3163	16552	3.24	512	PLRU	128KB
99.00%	1.00%	827198	5201	21764	3.52	512	RANDOM	256KB
90.00%	10.00%	925612	9536	40686	3.73	1024	LRU	64KB
97.00%	3.00%	121100	1489	7578	3.07	1024	PLRU	128KB
97.00%	3.00%	195285	2647	11078	3.14	1024	RANDOM	256KB
92.00%	2.00%	14706	547	3126	3.01	2048	PLRU	128KB
92.00%	8.00%	24790	1144	5496	3.03	2048	RANDOM	256KB

Fig. 11. With Prefetch

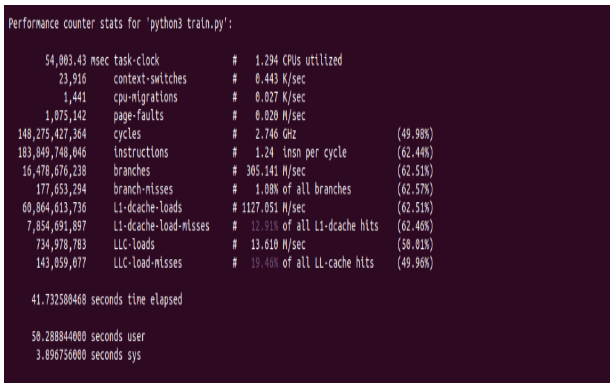
Experiments configurations with L2 and with prefetch and results can be seen in the below image.



Cache L1			With Prefetch					
Cache Size	R. Policy	Set Size	Miss Rate	Hit Rate	Access Count	Fetch Count	Prefetch Count	AMAT
16	LRU	1KB	5.00%	95.00%	4845651	258721	2046478	3.75
16	PLRU	32KB	2.00%	98.00%	4845651	105009	724114	3.24
16	RANDOM	128KB	5.00%	95.00%	4845651	225789	1202818	3.52
64	LRU	1KB	4.00%	96.00%	4845651	186581	1478062	3.73
64	PLRU	32KB	1.00%	99.00%	4845651	29357	183486	3.07
64	RANDOM	128KB	1.00%	99.00%	4845651	57645	275280	3.14
256	PLRU	32KB	1.00%	99.00%	4845651	3866	21680	3.01
256	RANDOM	128KB	0.20%	99.80%	4845651	8128	33324	3.03

Fig. 12. With prefetch

Following image shows the results of measuring performance of RNN using the perf tool.



```
Performance counter stats for 'python3 train.py':

      54,003.43 msec task-clock              #    1.294 CPUs utilized
         23,916      context-switches        #    0.443 K/sec
          1,441      cpu-migrations          #    0.027 K/sec
      1,075,142      page-faults             #    0.020 M/sec
148,275,427,364      cycles                  #    2.746 GHz                    (49.98%)
183,849,748,046      instructions            #    1.24  insn per cycle         (62.44%)
 16,478,676,238      branches                #  305.141 M/sec                  (62.51%)
    177,653,294      branch-misses           #    1.08% of all branches        (62.57%)
 60,864,613,736      L1-dcache-loads         # 1127.051 M/sec                  (62.51%)
  7,854,691,897      L1-dcache-load-misses   #   12.91% of all L1-dcache hits  (62.46%)
    734,978,783      LLC-loads               #   13.610 M/sec                  (50.01%)
    143,059,077      LLC-load-misses         #   19.46% of all LL-cache hits   (49.96%)

   41.732580468 seconds time elapsed

   50.288844000 seconds user
    3.896756000 seconds sys
```

Fig. 13. Performance Measure

Perf is used here to measure performance of the code in system. Perf tool is used for performance gathering like context switches between execution of the code, branches instructions cycles, stalled cycles are also gathered by perf tool.

IV. Results and Discussion

Different experiments were performed by varying cache memory size, block set number, block size, replacement policy, with and without L2 cache and prefetch. It is common that average memory access time of Recurrent neural network with Long Short-Term Memory with prefetch should be better then the results of without prefetch. It can be seen from the fig. 7 that all the experiments are performed without L2. Total access count of all the experiments are same but the fetch count varies; it can be seen that the fetch count while cache L1 size was 256Kb is lower then others. As it is discussed earlier that RNN code is used and then different tools are used for memory traces, heap consumption, load, store, memory and write back instructions.

In fig. 7, memory bar chart has been shown which was generated from the results that were achieved by the RNN code. By utilizing CVS documents information have been gathered by utilizing given order in a record of task and in this charts Instruction, Load, Store Memory have been thinking about what's more, the bar talk diagram is utilizing the power of these Data sets and assessing results on the graphical premise. In bytes, L, S, M and I are indicated which show that a most extreme number of directions executed are of I which is close around (500-11500) and afterward that of the store of various bytes, are around (3800-4000) and that of load (0-1200). So this bar graph shows that most extreme no. of instructions have been executed.

In Fig. 9 a table can been and these are the results that were achieved using the cache simulator. When L2 is missing, access time needs to be increased as the cache L2 is not available and the time has been increased as we have to fetch data from the main memory if there is a miss in cache L1. We saw here that with and without prefetch there is no as much difference between bringing fetch count, what's more, AMAT and hit rate is likewise practically the same since the reserve size is same just set no is being changed which tad change estimation of calculation.

It was observed that the results were good when prefetching is off although it should be the opposite i.e with prefetching results should be better then without prefetching. After observing the results and code in the details, it was seen that the RNN code, prefetching was not considered. Average memory access time with prefetching is higher then average memory access time without prefetching. There was a significant change noticed in access count and fetch count when size of cache increase because of size increment data scattered has been noticed and less main main memory access have to be performed.And Hit rate was also decreased for cache memory because most of the data is being found in L1 and L2 and that's why there was no need to travel to main memory for data access. That's why less memory access as well as hit rate in it have to be occur.

References

[1] Mikolov, Tom'a. Statistical language models based on neural networks. PhD thesis, Ph. D. thesis, Brno University of Technology, 2012.

[2] Graves, Alex, Mohamed, Abdel-rahman, and Hinton, Geoffrey. Speech recognition with deep recurrent neural networks. In Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, pp. 6645–6649. IEEE, 2013.

[3] Kalchbrenner, N. and Blunsom, P. Recurrent continuous translation models. In EMNLP, 2013.

[4] Mikolov, Tomas, Karafiat, Martin, Burget, Lukas, Cernock ´ y,‘ Jan, and Khudanpur, Sanjeev. Recurrent neural network based language model. In Interspeech, volume 2, pp. 3, 2010.

[5] Huang, Zhiheng, Geoffrey Zweig, and Benoit Dumoulin. ”Cache based recurrent neural network language model inference for first pass speech recognition.” In 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 6354-6358. IEEE, 2014.

[6] Peters, MaMark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. ”Deep contextualized word representations.” arXiv preprint arXiv.