Cache Replacement Policy for ML workloads

Rachit Tibrewal, Josh Mei

Introduction

This project evaluates cache replacement policies and other modifications to the Last Level Cache (LLC) for ML workloads on the ChampSim framework. Cache is a hardware component that stores frequently accessed data to speed up future requests. The cache replacement policy is an algorithm that determines which items to remove from the cache when it is full and new data needs to be stored. It aims to improve the performance of data access from the cache. The Cache Replacement Championship (CRC) is a competition to design the best performing cache replacement policy using the ChampSim microarchitectural simulator. The cache policy is designed for both private and shared LLC. These policies are evaluated on SPEC2006 and CloudSuite traces and with and without a data prefetcher. There is a fixed storage budget (32KB per core) for cache replacement policy implementation. All the policies will be evaluated under four configurations: single core with 2 MB LLC without a prefetcher, single core with 2 MB LLC with L1/L2 data prefetchers, a 4-cores with 8 MB of shared LLC without a prefetcher, a 4-cores with 8 MB of shared LLC with L1/L2 prefetchers.

Survey and Motivation

Our motivation for the project is to identify the effect of cache replacement policy on Machine Learning (ML) workloads. The paper "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective[4]" details Facebook's use of CPUs and GPUs for ML training and inference, with diurnal load cycles leaving numerous CPUs available for distributed training algorithms during off-peak periods, emphasizing the need for optimizing ML training and inference on CPUs. The significance of cache replacement policies in CPU design and their impact on system performance is critical, especially for Machine Learning inference and training workloads that process vast amounts of data in random or sequential order, requiring different cache replacement policies than general-purpose workloads to achieve optimal performance. Analyzing the cache replacement policies that perform best on CPUs can enhance resource utilization and guide the development of ML-specific workload policies, and studying the cache access pattern of ML workloads can reveal guiding principles for restructuring code.

A comprehensive survey of cache replacement policies can be found in the book “Cache Replacement Policies[6]” which summarizes the landscape of cache replacement policies by dividing policies into two categories, coarse-grained policies, and fine-grained policies, where each category is subdivided into multiple categories describing the different approaches to solve the cache replacement problem. Based on the categories, the authors describe the work with more consideration, conclude the trends and challenges for future work.

Hawkeye

One of the cache replacement policies we use is from the paper "Hawkeye: Leveraging Belady's Algorithm for Improved Cache Replacement[2]". The paper proposes cache replacement policies that are learned from Belady's optimal solution for past references to predict the caching behavior of future references. On the SPEC2006 workloads in the absence of prefetching the algorithms show a 4.5% speedup over Least Recently Used (LRU) and 3.4% speedup for SHiP. In the presence of prefetching, Hawkeye's performance is marginally better than SHiP's (2.25% speedup vs 2.09% speedup over LRU).

Hawkeye uses the OPTgen algorithm and to learn OPTgen's solution, it uses a PC-based predictor that learns whether load instructions tend to load cache-friendly or cache-averse lines. Lines predicted to be cache-friendly are inserted with high priority in the cache, while lines predicted to be cache-averse are inserted with low priority.

SHiP

The paper "SHiP: Signature-based Hit Predictor for High Performance Caching[7]" proposed a new cache replacement policy SHiP to predict the re-reference pattern of an incoming cache line, which will significantly improve the LLC's performance.

In our project, first we will use baseline algorithm LRU and SHiP to evaluate the cache replacement on specific ML workload.

- **LRU**: The commonly used policy which removes the least recently used item from a cache
- **SHiP**: The SHiP algorithm has a Signature History Counter Table (SHCT), SHCT indexed entry increments when cache line receives a hit; the indexed entry decrements when a line is evicted from the cache but has not been re-referenced since insertion. SHCT tracks the re-referenced frequency of the given signature other than its timing.

Then we use some algorithms from CRC to compare the performance with the baseline algorithms. We will pick SRRIP, Hawkeye (2nd Cache Replacement Championship winner) and SHiP++ policies for evaluation.

- **SRRIP**: LRU can be thought of as a Re-Reference Interval Prediction (RRIP) chain that represents the order in which blocks are predicted to be referenced. Static RRIP is a scan-resistant policy that requires only 2-bits per cache block and easily integrated into LRU policy[1].
- **Hawkeye**: Learned from Belady's optimal solution for past references to predict the caching behavior of future reference[2].
- **SHiP++**: To enhance the LRU policy, three areas of improvement can be focused on. These include modifying the policies for updating counters in the SHCT, utilizing a wider range of insertion points based on the SHCT counter value, and utilizing a separate SHCT table for demand and prefetch lines to improve cache performance during prefetching. These changes can result in a more efficient and effective LRU policy [3].

Test setup and survey on existing cache replacement policies

- **Simulator**: ChampSim

ChampSim is a trace-based simulator for a microarchitecture study. It supports branch predictor, data prefetchers and replacement policy simulation. In our project, we added the CRC (Cache Replacement Championship) replacement policies, created our own traces files from the ML workloads, and simulated on ChampSim.

- **Test device**: CSL machine

We use the CSL machine to run the evaluation tests, the cache hierarchy size is shown in the ChampSim configuration file.

Cache Level	Size	Latency
LEVEL1_DCACHE_SIZE	48 KB	5
LEVEL2_CACHE_SIZE	524 KB	10
LEVEL3_CACHE_SIZE	2MB per core	20

Table 1. ChampSim Cache Hierarchy

ML workloads traces and ChampSim simulation

We wrote programs for different Machine Learning and Deep Learning workloads and generated traces using the PIN tool. The program traces generated are uploaded to https://github.com/rachit173/ChampSim/tree/master/traces. Additional data regarding the simulations has been added to the sheet.

Matrix Multiplication

Matrix multiplication is the fundamental operation in machine learning algorithm, so we decided to create the traces based on matrix multiplications with the matrix dimensions large enough to hit the LLC. For a naive implementation of matrix multiplication, the working memory size is calculated as,

Working Memory Size = (3) * (N * N) * (4 bytes) = 12 * N * N * bytes
, where 3 is the number of matrices.

A	B	C = A x B	Working Memory size
(512, 512)	(512, 512)	(512, 512)	12 * 512 * 512 = 3 MB
(768, 768)	(768, 768)	(768, 768)	12 * 768 * 768 = 6.75 MB

Table 2. Matrix Multiplication Configs

Since the LLC is 2MB, the cache cannot store all three matrices simultaneously. Thus, LLC will fetch and replace data while running the program and use the cache replacement policy. We try the different cache replacement policy with default ChampSim configuration and record these results,

Policy	IPC	Total Access	Total Hit	Total Miss	LLC total miss rate
lru	0.122191	27371	10555	16816	0.614372876
lru with prefetch	0.122764	27990	10652	17338	0.619435513
srrip	0.122198	27371	10450	16921	0.618209053
srrip with prefetch	0.122782	27987	10547	17440	0.623146461
ship++	0.122194	27371	10554	16817	0.614409411
ship++ with prefetch	0.122768	27985	10629	17356	0.620189387
hawkeye	0.122194	27371	10580	16791	0.613459501
hawkeye with prefetch	0.122768	27985	10678	17307	0.618438449

Table 3. Matrix Multiplication Workload Simulation Results

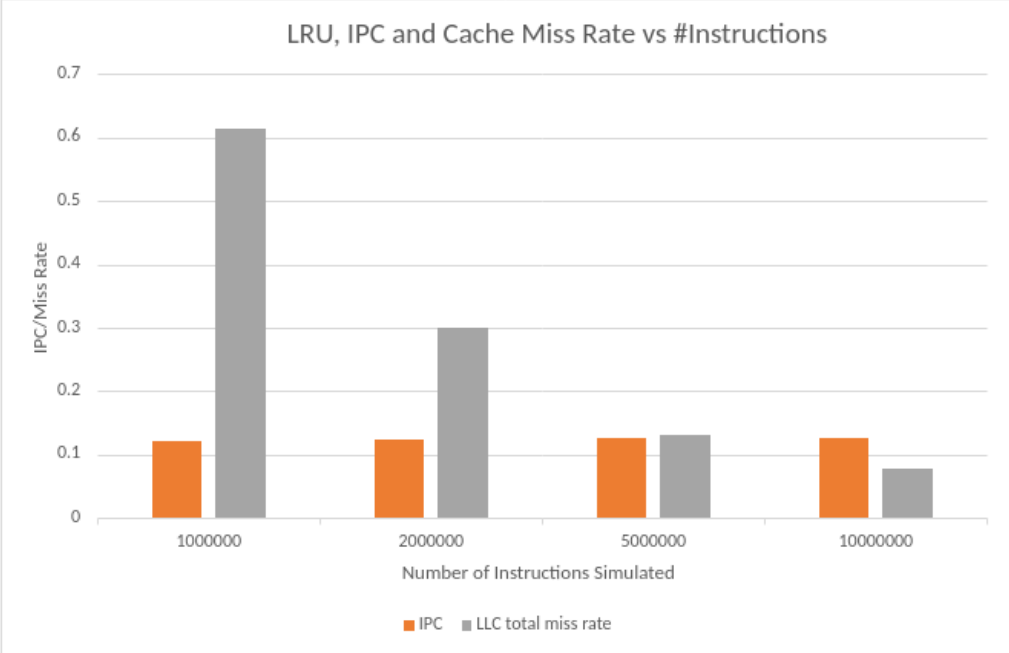

Figure 1. Matrix Multiplication Workload Under LRU Policy

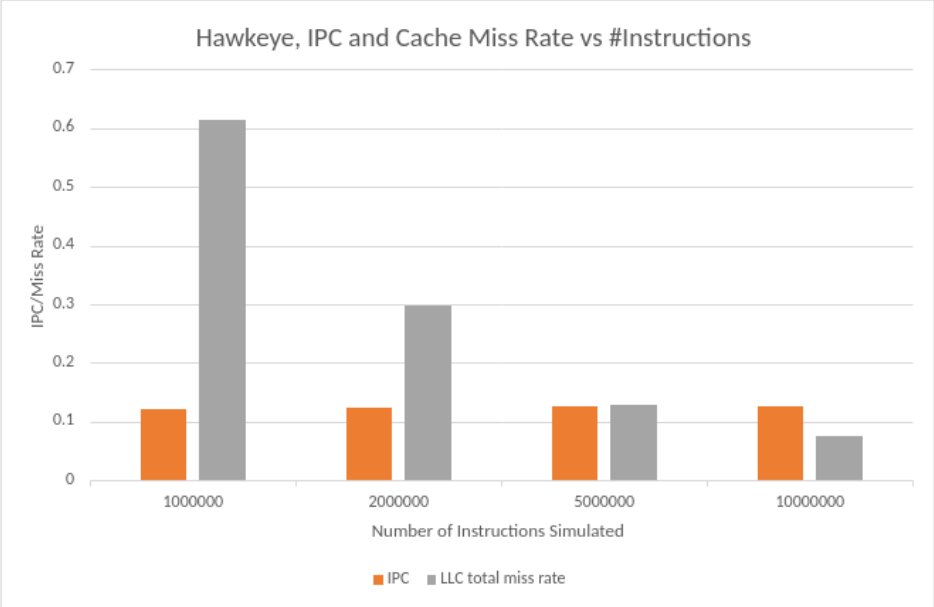

Figure 2. Matrix Multiplication Workload Under Hawkeye Policy

For the matrix multiplication workload, we observe the same IPC and cache hit rate for different cache replacement policies. The hit rate and IPC increases when prefetching is used. As the number of simulated instructions increases, the LLC cache miss rate decreases for both LRU and Hawkeye cache replacement policies. As more instructions are simulated, the cache is warmed up and the miss rate of the cache decreases.

PocketNN workload

PocketNN is a framework that enables training and inference of deep neural networks (DNNs) on low-end devices using only integers. It uses pocket activations and direct feedback alignment (DFA) to operate directly on integers, avoiding the need for explicit quantization algorithms or fixed-point formats.

Policy	IPC	Total Access	Total Hit	Total Miss	LLC total miss rate
lru	0.499031	7338	617	6721	0.915917144
lru with prefetch	0.555724	8395	792	7603	0.90565813
srrip	0.499031	7338	617	6721	0.915917144
srrip with prefetch	0.555724	8395	792	7603	0.90565813
ship++	0.499031	7338	617	6721	0.915917144
ship++ with prefetch	0.555724	8395	792	7603	0.90565813
hawkeye	0.499031	7338	617	6721	0.915917144
hawkeye with prefetch	0.555724	8395	792	7603	0.90565813

Table 4. PocketNN Workload Simulation Results

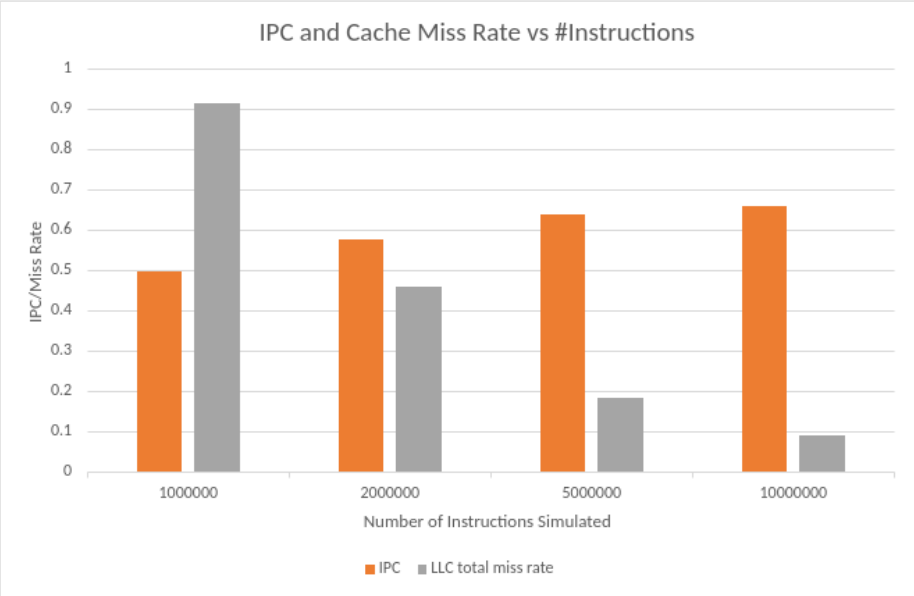

Figure 3. IPC and Cache Miss Rate in PocketNN Workload

For PocketNN workload, we observe no difference in cache miss rate and IPC across different cache replacement algorithms. When prefetching is used with the cache replacement algorithms, the LLC miss rate improves by 1% and the IPC increases by 10%. From Figure 3, we can observe that as more instructions are simulated, the IPC increases, and the total miss rate decreases significantly. Since the PocketNN model and workload are smaller than the LLC, after one epoch of training the entire model and dataset will be loaded in the LLC. Thus, subsequent access to model and dataset will not result in misses.

LeNet-5 with MNIST dataset workload

LeNet is a CNN architecture developed for handwritten digit recognition. MNIST is a large database of handwritten digits used for training and testing image processing systems and is widely considered a benchmark dataset for image classification. We have created a ML workload trace using LeNet with MNIST dataset.

Policy	**IPC**	**Total Access**	**Total Hit**	**Total Miss**	**LLC total miss rate**
lru	0.4496	9371	1143	8228	0.878027959
lru with prefetch	0.49759	11761	1615	10146	0.862681745
srrip	0.4496	9371	1143	8228	0.878027959
srrip with prefetch	0.49759	11761	1615	10146	0.862681745
ship++	0.4496	9371	1143	8228	0.878027959
ship++ with prefetch	0.49759	11761	1615	10146	0.862681745
hawkeye	0.4496	9371	1143	8228	0.878027959
hawkeye with prefetch	0.49759	11761	1615	10146	0.862681745

Table 5. LeNet-5 with MNIST Dataset Workload Simulation Results

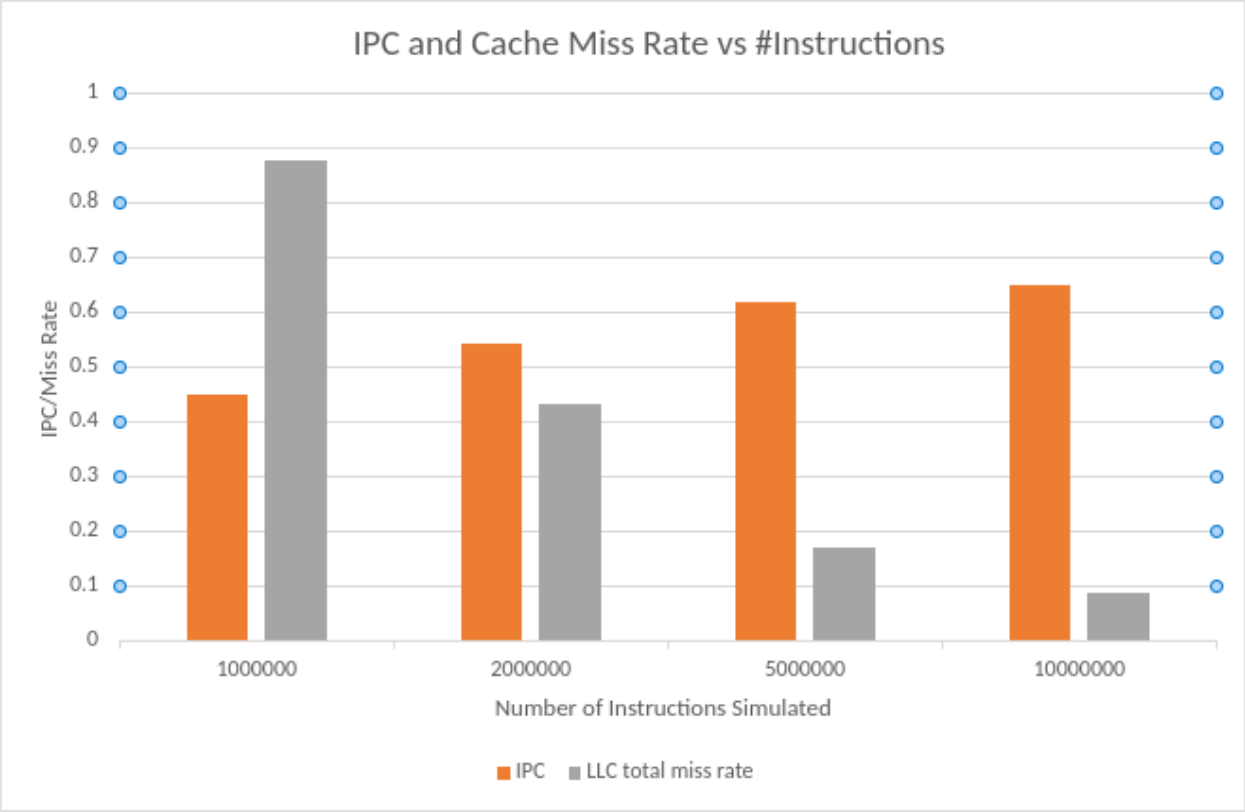

Figure 4. IPC and Cache Miss Rate in LeNet-5 with MNIST Dataset Workload

Similar to PocketNN model with MNIST, LeNet-5 model with MNIST shows no difference in performance for different cache replacement algorithms. When prefetching is used with the cache replacement algorithms, the LLC miss rate improves by 1% and the IPC increases by 10.7%. From Figure 4, we can observe that as more instructions are simulated, the IPC increases, and the total miss rate decreases significantly. Thus, subsequent access to model and dataset will not result in misses.

Inference of ResNet-18 with random inputs workload

ResNet-18 is a deep residual neural network architecture consisting of 18 layers, widely used for image recognition tasks due to its excellent performance and ability to avoid the vanishing gradient problem. We evaluate ResNet-18 on dummy batches of 128 images where each image is 224x224. The dummy input does not affect the performance of the model. We evaluate the model on 10 batches of data. ResNet-18 has 11M parameters which is approximately 44 MB in size. This is significantly larger than LLC.

Policy	IPC	Total Access	Total Hit	Total Miss	LLC total miss rate
lru	0.453022	9423	1233	8190	0.869149952
lru with prefetch	0.500706	11734	1637	10097	0.860490881
srrip	0.453022	9423	1233	8190	0.869149952
srrip with prefetch	0.500706	11734	1637	10097	0.860490881
ship++	0.453022	9423	1233	8190	0.869149952
ship++ with prefetch	0.500706	11734	1637	10097	0.860490881
hawkeye	0.453022	9423	1233	8190	0.869149952
hawkeye with prefetch	0.500706	11734	1637	10097	0.860490881

Table 6. ResNet-18 with Random Inputs 1 Batches Workload Simulation Results

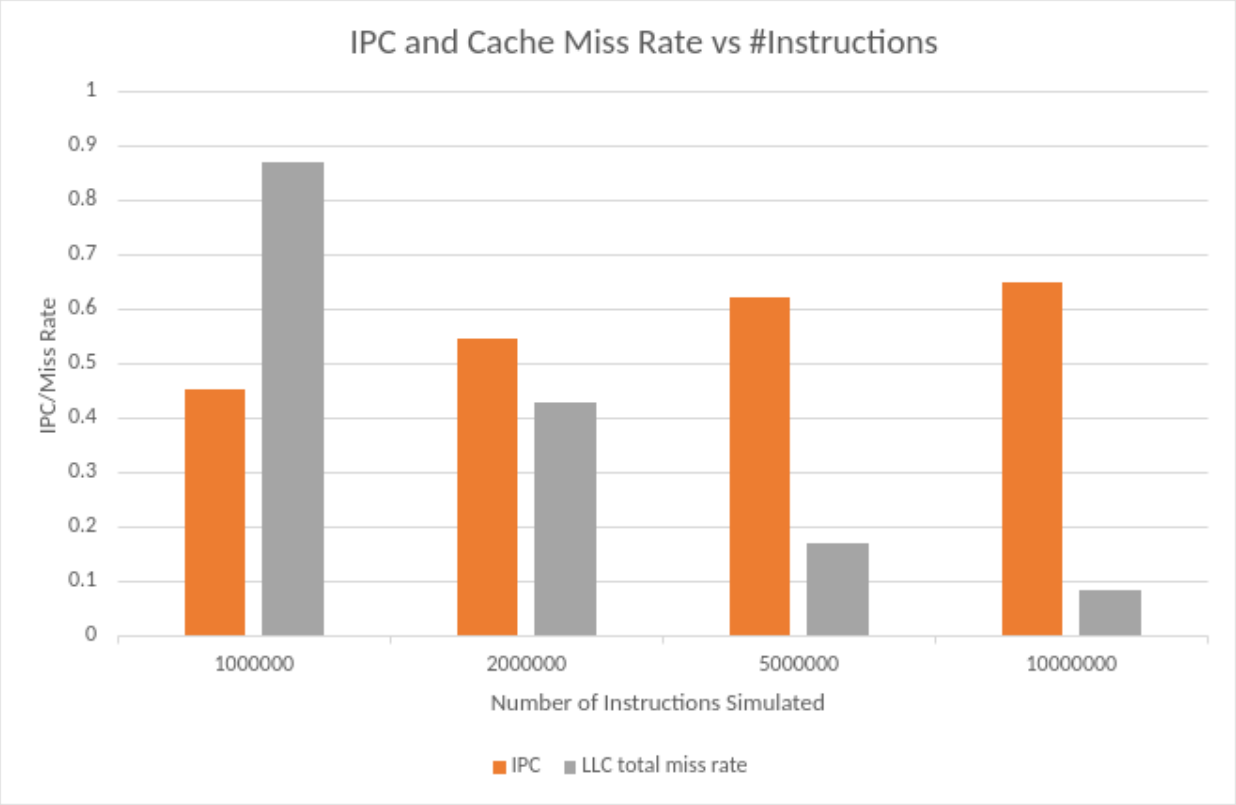

Figure 5. IPC and Cache Miss Rate in ResNet-18 with Random Inputs 1 Batches Workload

Policy	IPC	Total Access	Total Hit	Total Miss	LLC total miss rate
lru	0.453518	9374	1170	8204	0.875186687
lru with prefetch	0.500761	11746	1631	10115	0.861144219
srrip	0.453518	9374	1170	8204	0.875186687
srrip with prefetch	0.500761	11746	1631	10115	0.861144219
ship++	0.453518	9374	1170	8204	0.875186687
ship++ with prefetch	0.500761	11746	1631	10115	0.861144219
hawkeye	0.453518	9374	1170	8204	0.875186687
hawkeye with prefetch	0.500761	11746	1631	10115	0.861144219

Table 7. ResNet-18 with Random Inputs 10 Batches Workload Simulation Results

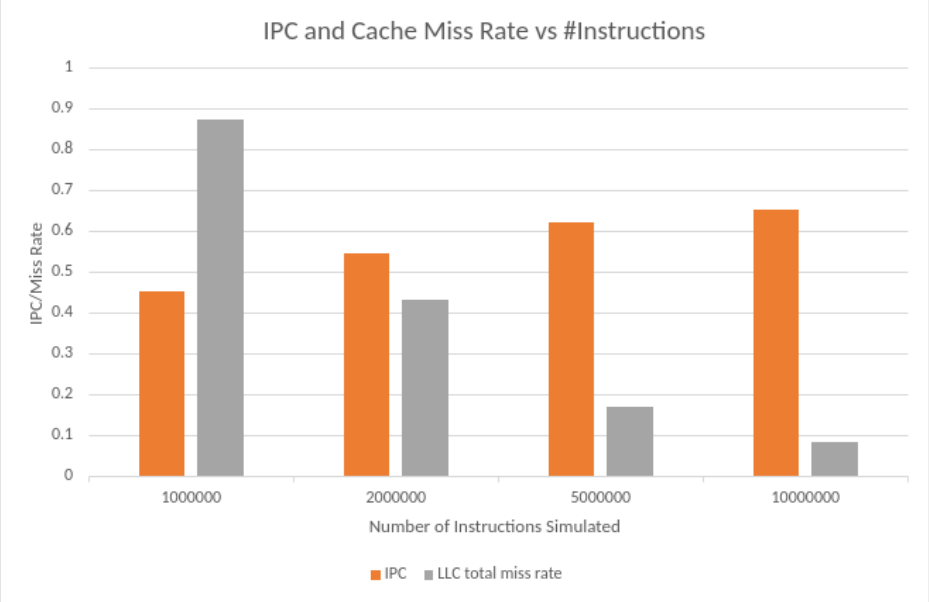

Figure 6. IPC and Cache Miss Rate in ResNet-18 with Random Inputs 10 Batches Workload

For ResNet-18 model, we get similar performance of model for different cache replacement policies. Prefetching leads to a 10% improvement in IPC. Unlike LeNet-5 where the model can completely fit in the LLC, for ResNet-18, parts of the model will be fetched by the LLC repeatedly. When some consecutive layers are fetched to the LLC, it will completely fill the LLC leaving no room for other layers. When subsequent layers are fetched for inference, they will displace the previous layers in cache. For the inference of a new batch of data, all the layers of the model will need to be fetched from the main memory, resulting in cache misses for the blocks of data. Since the model parameters are placed sequentially in memory, prefetching helps reduce cache misses by fetching more data from the memory in every fetch.

Inference of BART on text filling task

We use BART [8] for a text filling task. An example of text filling task is where a sentence “UN Chief Says There Is No <mask> in Syria” is passed to the model and the model replaces the mask to generate an output sentence like "UN Chief Says There Is No Plan to Stop Chemical Weapons in Syria". BART is a large language model built on the transformer architecture. These models are very effective on many natural language processing tasks. The basic building block of large language model is the attention block. Matrix multiplication is the most computationally expensive operation in the attention block. Several attention blocks are stacked on top of each other and once computation at a layer is complete the model weights of that layer are not required in the subsequent layers. The total size of model parameters in BART is 1.5 GB which is significantly larger than the LLC.

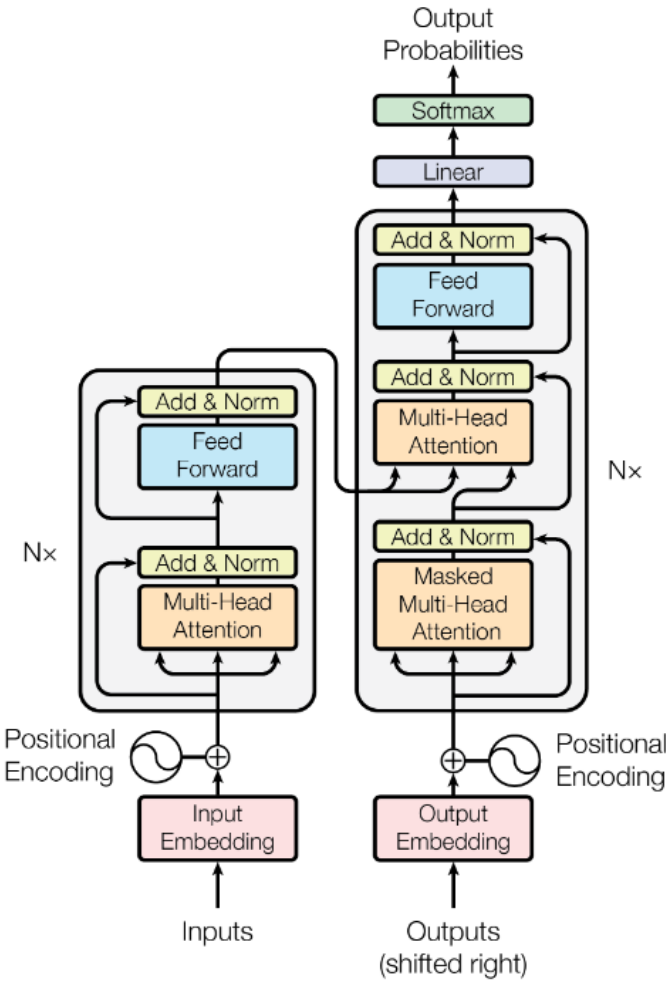

Figure 7. Transformer Model Architecture[9]

Policy	IPC	Total Access	Total Hit	Total Miss	LLC total miss rate
lru	0.318111	22353	5445	16908	0.756408536
lru with prefetch	0.341311	23601	5959	17642	0.747510699
srrip	0.318111	22353	5445	16908	0.756408536
srrip with prefetch	0.341311	23601	5959	17642	0.747510699
ship++	0.318111	22353	5445	16908	0.756408536
ship++ with prefetch	0.341311	23601	5959	17642	0.747510699
hawkeye	0.318111	22353	5445	16908	0.756408536
hawkeye with prefetch	0.341311	23601	5959	17642	0.747510699

Table 7. ResNet-18 with Random Inputs 10 Batches Workload Simulation Results

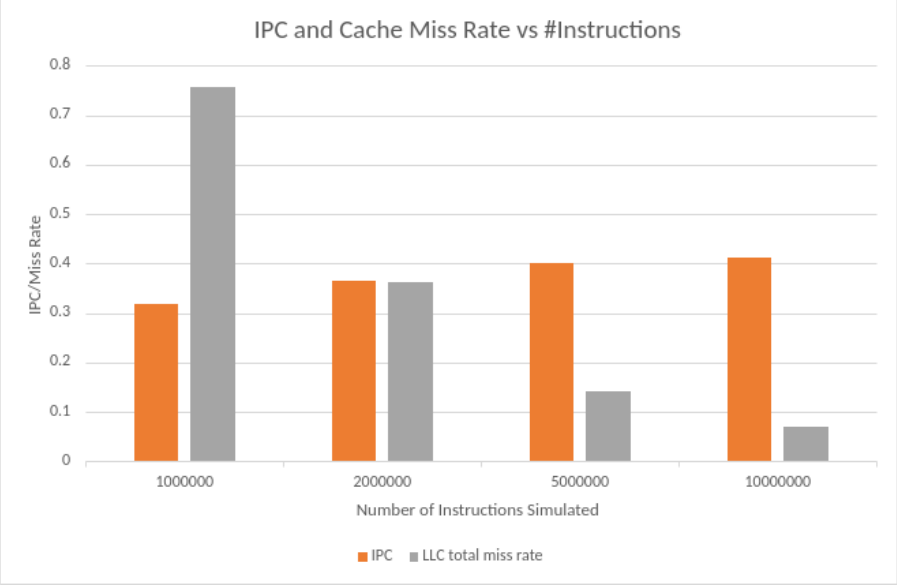

Figure 8. IPC and Cache Miss Rate in BART Workload

Using different cache replacement policies does not change the performance on the BART workload. Prefetching with different cache replacement policies results in 7% improvement in IPC. BART is a very large deep learning model (1.5 GB), and only small parts of the model parameters can be loaded to the LLC at a time. The inputs for BART model are typically less than a KB in size. The model inference is done layer by layer, and when a new layer gets loaded in the LLC, the previous layer gets evicted since the layers are larger than the LLC. Thus, when inference of a new batch of data is performed, the model layers need to be fetched from main

memory again as no relevant model parameters remain in the LLC. Different cache replacement policies will have no effect on the performance of LLC as there is no temporal locality in the data access.

In addition, we investigated that with different LLC configurations, where we keep the cache size the same but change the sets and ways size, the IPC and miss rate both showing different results. With the increase of cache ways and the decrease of cache sets, the LLC IPC increases, the miss rate decreases, which means that higher LLC associativity improves the data access performance for BART workload.

Config	LLC Cache Size	sets	ways	block size
Config 1	2097152	8192	4	64
Config 2	2097152	4096	8	64
Config 3	2097152	2048	16	64
Config 4	2097152	1024	32	64

Table 8. Four Configurations for Cache Sets and Ways in BART Workload Simulation

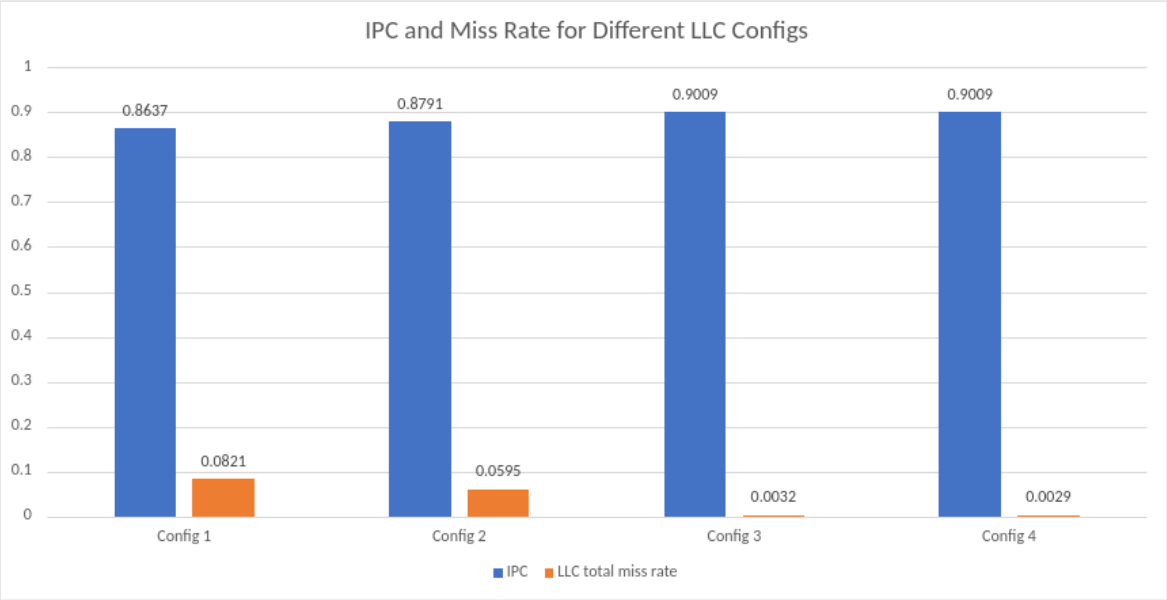

Figure 9. IPC and Miss Rate for Different LLC Configurations in BART Workload Simulation

OBJ

Conclusions

The machine learning models used for regression tasks and computer vision are smaller compared to the size of LLC. Thus, after one epoch of training or a batch of inference, the model is loaded in the LLC and all subsequent accesses to model parameters will result in hits. For larger models like BART, the model parameters cannot fit in the LLC. Typically, the parameters of the layer for which the computation is taking place will completely fill the cache, leaving no room for model parameters of previous layers. Overall, we find that there is no benefit in using a different cache replacement policy than LRU policy for the LLC for ML workloads.

Using prefetcher with the cache replacement policy for the LLC results in performance gains. For different workloads we observe a reduction in cache miss rate of 1% and improvement in the IPC ranging from 7%-11%. This gain can be observed across algorithms and different cache replacement policies.

We also study the effect of varying the associativity of the LLC while keeping the cache size constant. Increasing the cache associativity results in a significant decrease in miss rate and increase in the IPC. This study suggests using a large set-associativity for the LLC with Machine Learning workloads as it improves cache performance significantly.

References

[1] A. Jaleel, K. B. Theobald, S. C. Steely Jr. and J. Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). ISCA'10, Saint-Malo, France, 2010.

[2] A. Jain and K. Lin. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, Korea, 2016.

[3] V. Young, C. Chou, A. Jaleel and M. Qureshi. SHiP++: Enhancing Signature-Based Hit Predictor for Improved Cache Performance. The 2nd Cache Replacement Championship, co-located with ISCA, 2017.

[4] K. Hazelwood, S. Bird, D. Brooks, et al. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. International Symposium on High-Performance Computer Architecture (HPCA), 2018.

[5] J. Song and F. Lin. PocketNN: Integer-only Training and Inference of Neural Networks via Direct Feedback Alignment and Pocket Activations in Pure C++. tinyML Research Symposium, ACM, New York, USA, 2022.

[6] A. Jain and K. Lin. Cache Replacement Policies. Morgan & Claypool, 2019.

[7] C. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely and J. Emer. SHiP: Signature-based Hit Predictor for High Performance Caching. 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, 2011.

[8] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, Omer Levy, Veselin Stoyanov, Luke Zettlemoyer. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin. Attention Is All You Need. 31st Conference on Neural Information Processing Systems, Long Beach, USA, 2017.