Homework 2: Building Neural Models for the Cache Replacement Problem

Kaushal Mhapsekar, AJ Beckmann, Jakub Jon, Anjolie Baccay
Department of Electrical and Computer Engineering, NC State University

***Abstract*—This report builds on Homework-1, which presents a comprehensive study into the PARROT cache replacement policy [10]. Team *AJAK* dives into space exploration & experimentation with the PARROT model and ChampSim simulator to prepare data for a RAG. We then build an LLM application to test its ability as a "cache replacement policy" and to get insights for improving the existing MLP and RNN models. As part of our work, we are releasing all of our code on Github under the name *RAGGEN: Slangin' data and shootin' the RAG*.[1]**

***Index Terms*—Imitation learning, LSTM, attention mechanism, cache replacement policy, retrieval-augmented generation (RAG), large language model (LLM)**

I. Introduction

In Homework-1 the members of this team aimed to reproduce the results generated by the PARROT cache replacement model created in [10]. This model is representative of a growing influx of implementing AI models in microprocessors to further optimize their efficiency and performance. These applications range from branch prediction [4] [6], to reuse prediction [7] [11], to the cache replacement applications [5] [13] seen in the PARROT [10] and Glider [14] models. All of these have shown promising results in improving their respective tasks over existing non-ML based solutions. However, what inhibits the wide acceptance of these models in practice is their unproven trustworthiness. The National Institute of Standards and Technology highlights four characteristics of trustworthy AI that are of interest to this paper: accountability, transparency, explainability, and interpretability. Essentially, we often lack the conceptual insight into how these models are achieving the results they produce.

Credibility for humans is derived from reasoning and logical comprehension, where results are traceable back to the fundamental concepts they are built off. We expect, and even demand, the same from AI models. This heavily translates to the ability to conceptualize the features and intermediate processes of a model and how inferences are logically constructed from them. However, with many complex models, including the model explored in this paper, this is often obscure.

Therefore, in this paper we aim to explore the features and processes that the PARROT model utilizes when developing its cache eviction policy. We employ a bottom-up approach by changing and implementing low-level processes throughout the model and analyzing how they affect the results. Additionally, we aim to supplement the state-of-the-art by implementing a retrieval-augmented generation (RAG) system to help develop reasoning and insight into PARROT's execution.

[1]https://github.com/lnotspotl/raggen

II. Background

The problem of cache replacement and its formulation as a imitation learning as the PARROT eviction policy is discussed in [10]. Homework-1 involved setting up and simulating the results generated by PARROT. A report for Homework-1 can be found in Homework-1 Report.

A Retrieval-Augmented Generation (RAG) system is a data retrieval technique used in collaboration with large language models (LLMs) to improve an LLMs reasoning and insight into a large amount of information. In a RAG system, an LLM is augmented by an external knowledge base, such as a collection of documents or a database, to retrieve relevant information and incorporate it into the output generation process. RAG systems have been shown to enhance an LLM's ability to answer complex questions or generate contextualized and accurate outputs on topics beyond its training data [9].

III. Tasks

This section presents the details of the work done by our team on each task. We follow the format of highlighting the objective of the task performed, methodology followed during experimentation, results from the task, discussions on the results, challenges faced and ideas presented in the future work we plan to do for the final project.

A. Task-0

Objective: This task aims to investigate into the errors encountered during Homework-1, find possible solutions to them, and formulate a good methodology and metrics for Homework-2 and Final Project.

Methodology: To investigate the error we first need to understand the process we are following in Homework-1. The following steps were followed in Homework-1 as given in the GitHub repository for the PARROT Paper:

1) Download all dependencies and setup a virtual environment.
2) Git clone ChampSim and git checkout to an older branch version of ChampSim
3) Git apply the patch file provided by the authors
4) Build and run ChampSim simulator to get "llc_access_trace.csv"
5) Split the generated LLC access trace file for the chosen benchmark into train, valid and test sets. (80-10-10 split)

6) Simulate cache usage for the benchmark using the baseline replacement policies provided by the authors (PARROT, LRU, Belady's).
7) Evaluate the policy and compare the results

To investigate further, we cloned and simulated the same benchmarks on the latest version of ChampSim. We made a crucial observation that after 1 billion instructions, the simulator loops the trace and simulates over the same 1 billion instructions again. This observation is strange considering that the benchmarks are supposed to have a significantly more number of instructions based on the filename (eg: astar_313B, omnetpp_4B,etc).

The authors of PARROT make changes to ChampSim source code to not perform this looping of the trace using the patch file. This is done to not have an overlap of the training and testing datasets. The "core dump" error only occurs when the number of simulation instructions is more than 1 billion. This points to the possibility that the core dump error which is occurring due to segmentation fault, is likely caused due to the simulator trying to open a file that is not longer than 1 billion instructions.

Discussion: Based on the observations made about core dump error, we have replied to an Open GitHub Issue[2], hoping to help back the open source community stuck on the same error.

Future Work: There still needs to be further investigation into why we all were unable to exactly replicate the results of the PARROT paper during Homework 1. Based on some of the combined observations made by the team, we were able to get results close to the authors for some of the benchmarks. Specifically looking at the results of astar, we were able to get raw hit rates which are similar to the authors for one of the three workloads (astar_313B). The class trained and evaluated the model on the same trace files that the class assistant had extracted, which could be the reason why everyone got similar results to each other but not the authors. It could be possible that we did not train and evaluate PARROT on the exact workloads the authors used. Therefore, for the Final Project, we would also like to train the PARROT model on all the 3 workloads (Number of instructions) of every benchmark, to confirm this hypothesis and try to successfully replicate the results from PARROT.

B. Task-1

Objective: This task aims to compare the MPKI metric, sort, and select different benchmarks based on highest MPKI values for the experimentation for the rest of the homework.

Methodology: Misses per Kilo Instructions (MPKI) is a standard metric used in measuring Microarchitecture Performance and is given by the formula:

$$MPKI = \frac{Total Number of misses}{Total Number of instructions} \times 1000$$

This is a useful metric to compare different benchmarks, because we are interested in the performance, which is measured by the number of misses, while keeping the comparison of benchmarks fair by normalizing it to number of instructions. It is important to understand the difference between cache miss rate and misses per kilo instructions. Cache miss rate is given by:

$$MissRate = \frac{Total Number of misses}{Total Number of Cache Accesses}$$

Cache Miss Rate tells us the efficiency or effectiveness of the cache under consideration, whereas, MPKI provides insights into the impact of cache misses onto the performance of CPU.

With that background, we will use MPKI as our primary metric for evaluation of different tasks but would also provide cache miss rates to derive a deeper understanding. There are two approaches to get the MPKI values of a certain benchmark using a certain Cache Replacement Policy:

1) Simulating the traces on ChampSim.
2) Using the results of evaluation on PARROT infrastructure.

We are evaluating PARROT and our experimental neural models of these tasks on the "test.csv" files for every benchmark. This has to be done to not train and test on the same data. The number of misses thus reported are due to a tenth of the LLC access trace generated for the benchmarks. The authors also sample a fraction of the cache sets to reduce the model size. These factors would lead to slight variation or inconsistencies in the MPKI values.

For method-1, we can use ChampSim to simulate the number of misses and the number of instructions and then calculate MPKI using the formula. For method-2, we get the number of misses from the misses.csv file generated during evaluation and multiply it by 10 to get a rough value of number of misses for the complete LLC access trace. PARROT, samples 64 cache sets out of the 2048 total sets present in the LLC to train and test. Taking this into account, we also need to scale the number of misses by a factor of 204864 to get the total number of misses. We then use the same formula to calculate MPKI values.

$$Total Number of Misses = \text{m} \times 10 \times \frac{2048}{64}$$

where m is the number of misses in test.csv file of the benchmark.

For task-7 requiring comparison with other replacement policies, we used ChampSim and the replacement policies provided by them. For other tasks we have used the PARROT infrastructure and method 2 to calculate MPKI values. Therefore, for this task we have used both ChampSim and PARROT Infrastructure to get the MPKI values.

Note, we are working with LLC and measuring & improving its performance, therefore, our metrics look only at the LLC and not L1 and L2 caches.

Results: Table-I, shows the statistics for LRU replacement policy on ChampSim. Table-II, shows the calculation of MPKI values for LRU replacement policy on PARROT Infrastructure. Table-III, shows the calculation of MPKI values for the PARROT replacement policy. The results for these tables are plotted in Fig. 1 & Fig. 2

Discussion:

[2]https://github.com/google-research/google-research/issues/683

Trace	Total # of misses	# of cache accesses	MPKI	Miss Rate	Hit Rate
mcf_250B	92121410	131255840	92.12141	0.7018	0.2982
astar_313B	53321610	84370420	53.32161	0.6320	0.3680
lbm_564B	30530548	50991280	30.53055	0.5987	0.4013
milc_409B	26932302	36448944	26.93230	0.7389	0.2611
GemsFDTD_712B	24305532	36012729	24.30553	0.6749	0.3251
omnetpp_4B	15819149	30936897	15.81915	0.5113	0.4887
sphinx3_1339B	12730874	15009153	12.73087	0.8482	0.1518
astar_23B	2313450	14838825	2.31345	0.1559	0.8441
astar_163B	2304734	4290051	2.30473	0.5372	0.4628

TABLE I: LLC Cache Statistics simulated from ChampSim

Trace	# of misses (test)	Total # of misses	MPKI
mcf_250B	273703	87584960	87.58496
astar_313B	134566	43061120	43.06112
lbm_564B	95424	30535680	30.53568
milc_409B	91169	29174080	29.17408
omnetpp_4B	43623	13959360	13.95936

TABLE II: Calculation of MPKI for LRU policy using PARROT infrastructure

Trace	# of misses (test)	Total # of misses	MPKI
lbm_564B	95424	30535680	30.53568
astar_313B	94178	30136960	30.13696
milc_409B	91172	29175040	29.17504
leslie3d_1116B	86600	27712000	27.71200
GemsFDTD_712B	79213	25348160	25.34816
sphinx3_1339B	20856	6673920	6.67392
xalancbmk_748B	9873	3159360	3.15936
astar_163B	6184	1978880	1.97888
cactusADM_1039B	2290	732800	0.73280

TABLE III: Calculation of MPKI for PARROT policy using PARROT infrastructure

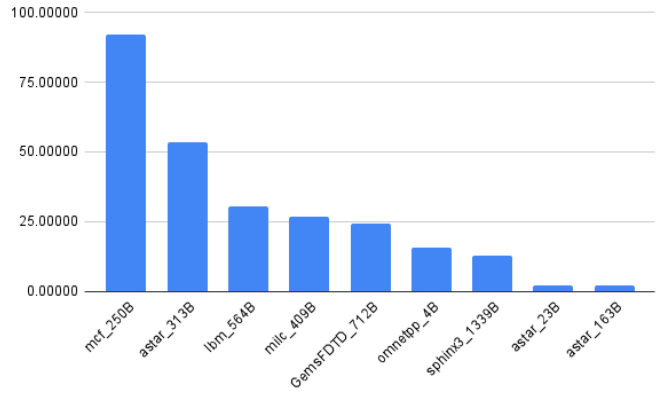

Fig. 1: MPKI for SPEC benchmarks on ChampSim using LRU policy

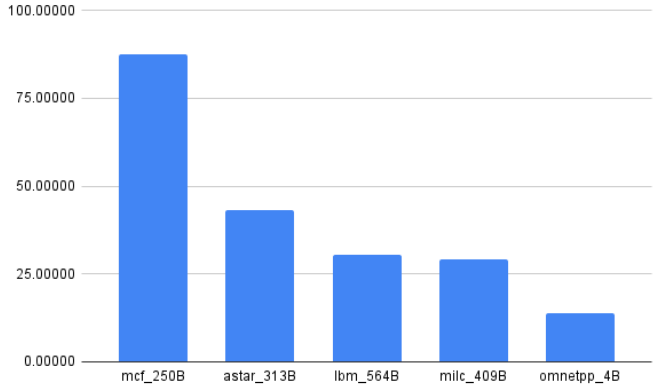

Fig. 2: MPKI for SPEC benchmarks on PARROT infrastructure using LRU policy

Comparing the MPKI values from Table-I & Table-II, we can see that the top chosen benchmarks have negligible difference in their values and follow the same trend.

Based on all the results for LRU across both methods, we shortlisted five benchmarks to run our experiments on - astar_313B, lbm_564B, milc_409B, mcf_250B and omnetpp_4B. Although the primary reason for these choices is the MPKI value, we have also considered factors like miss rate, comparison with Glider and nature of programs (memory-intensive benchmarks) while selecting the benchmarks.

Challenges: Possible inconsistency in results and mismatch with results from ChampSim.

Future Work: Include Normalized MPKI results for all experiments. [8]

C. Task-2

Objective: Task-2 consists of ablating the LSTM and attention mechanism from the PARROT cache eviction model and purely using a single linear layer architecture for cache replacement.

Methodology: Having only a single linear layer means that the entire eviction policy will depend on the connections formed in this layer. However, the same information, namely the cache state $\mathbf{s}_t = (\mathbf{s}_t^c, \mathbf{s}_t^a, \mathbf{s}_t^h)$, is still available to the linear layer at its input. The difficulty then becomes presenting the cache state information to the linear layer in a digestible manner that allows relevant relationships to be developed

between the current cache access and the available cache lines. Another challenge lies in ensuring that the change of the model architecture does not result in incompatibility with the rest of the system processes, such as loss calculation and training iteration. As the code base is extensive and complex and has many widely distributed dependencies, changes to the model must be tactfully done as to allow smooth operation of the full system.

To tackle both of these problems, a specific input structure was constructed. This structure, shown in Fig. 3, was chosen to maintain the same form that the output of the attention mechanism produces, namely, the python variable `context` . This input structure has the dimensions (batch size$\times$#cache lines$\times$ $(2d_m + 2d_{\text{pc}})$) where d_m is the embedding length of the address embedder and d_{pc} is the embedding length of the pc embedder. For the normal training cycle this is usually $(32 \times 16 \times (128 + 128)$. For a given memory access, its pc and address embeddings are concatenated and copied #cache lines times where they are then concatenated with each cache line's pc and address embedding. When three dimensional inputs are given to a PyTorch linear layer, the model takes the last dimension as its input and considers the multiplication of the first two dimensions as separate samples. This means that for each cache access, the linear layer will perform 16 computations, where each computation takes the cache access embedding and one of the 16 cache line embeddings. We recognize that this is still an incomplete representation of the information available for a given cache access, as a given input only contains information of a single cache line. However, the thought process is that, since the loss and backward propagation is performed on the total loss of a whole batch, the model will be optimized after seeing all cache lines for a given cache access rather than optimizing after seeing the cache access and one of its cache lines.

The output of the single linear layer is of dimensions (batch size $\times$ #cache lines), which is equivalent to the size of the output generated by the last linear layer of the full PARROT model, which allowed this single layer model to be easily compatible with the rest of the model processes.

Implementing these architectural changes mainly took place in main.py and model.py.

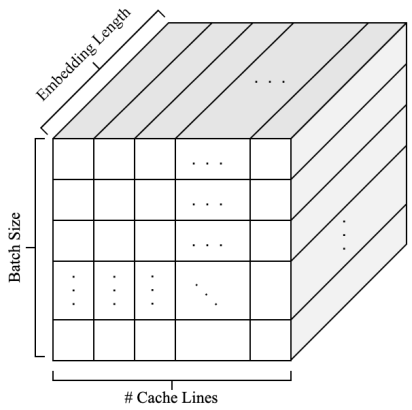

Fig. 3: Single and multi-layer perceptron input format.

Results: The Task-2 model was evaluated at the 68,000 step checkpoint for each of the five workloads designated in Task-1. As each workload evaluates to a different number of steps due to their differing number of instructions, a common step of 80,000 was used for analyzing the model's performance. These results for each workload are summarized in TABLE IV.

Benchmark	MPKI	Raw Cache Hit Rates	Normalized Cache Hit Rates
astar	29.9802	0.3164	0.8095
lbm	24.2979	0.2094	0.7822
mcf	67.2883	0.4401	0.8586
milc	28.5434	0.02987	0.5166
omnetpp	3.9469	0.8553	0.9131

TABLE IV: Task-2 Evaluation Metrics

D. Task-3

Objective Task-3 consists of ablating the LSTM and attention mechanism from the PARROT cache eviction model and using a two layer perceptron architecture for cache replacement.

Methodology: Task-3 was structured similarly to Task-2, with the addition of one more linear layer. The same input structure discussed in Task-2 is implemented in Task-3, with the key difference being that the output of the first linear layer maintains the same size as the input. The second linear layer is chosen to be the one already implemented in the full PARROT model. This layer, called the `cache_line_scorer`, originally takes the `context` variable as input. In Task-3, however, this input is replaced by the output of the first linear layer, producing an output of size (batch size $\times$ #cache lines).

Results: The Task-3 model was evaluated at the 68,000 step checkpoint for each of the five workloads designated in Task-1. Again, the 80,000 step evaluation metrics were analyzed. These results are summarized in TABLE V.

Benchmark	MPKI	Raw Cache Hit Rates	Normalized Cache Hit Rates
astar	30.0115	0.3318	0.8555
lbm	24.3606	0.2094	0.7822
mcf	67.3443	0.4401	0.8586
milc	28.3456	0.02987	0.5166
omnetpp	3.9456	0.8553	0.9131

TABLE V: Task-3 Evaluation Metrics

Task-2 & Task-3 Challenges: Implementation of Tasks 2 and 3 were tricky, as the model execution is sensitive to changes in the dimensions of inputs and outputs of certain layers. Specifically, without needing substantial changes to the main execution of the model, the `cache_line_scorer` layer, `reuse_distance_estimator` layer, loss and metric calculation process, and training and evaluation processes expect an output dimension of (batch size $\times$ #cache lines). Furthermore, this dimension can not be hard coded, as the model passes in different size inputs when training and evaluating. A different input structure implementation other than the one mentioned above was considered in an attempt to

encompass all of the information that the original model would have seen, namely, the cache access, history, and all the cache lines in the cache. The attempt involved concatenating the PC of the cache access, the PCs of n past cache accesses, and the PC of each each line in the cache. However, this would inherently involve defining the number of cache lines when defining the size of the input to the linear layer at the initialization of the model. As the number of cache lines is not always consistent and dynamically sized linear layers are extremely rigorous to implement in PyTorch, this method was unachievable. Reformation of this method eventually led to the input structure that was implemented above.

E. Task-4

Objective: Task-4 involves replacing the LSTM in the original PARROT model with an RNN, while maintaining all others aspects of the model.

Methodology: The implementation of Task-4 was quite simple as the functionality of and LSTM a RNN are similar. The original PARROT model uses a PyTorch LSTM cell layer like so:

```
self.lstm = nn.LSTMCell()
next_c, next_h = self.lstm()
```

Task-4 was implemented by replacing the LSTM cell with a RNN cell layer like so:

```
self._rnn_cell = nn.RNNCell()
next_h = self._rnn_cell()
```

The RNN cell function does not return a cell state, so this was eliminated from the outputs and set to `null` when referenced elsewhere in the code base.

Results: The Task-4 model was evaluated at the 68,000 step checkpoint for each of the five workloads designated in Task-1. The 80,000 step results are summarized in TABLE VI.

Benchmark	MPKI	Raw Cache Hit Rates	Normalized Cache Hit Rates
astar	30.232	0.3183	0.8151
lbm	30.536	0	0
mcf	25.122	0.2121	-0.3907
milc	29.172	0.00125	0.0009
omnetpp	5.0595	0.8134	0.8077

TABLE VI: Task-4 Evaluation Metrics

Challenges: Task-4 proved challenging as it was extremely slow when training and evaluating. Given a training time of 24 hours on the mcf_250B workload (which usually took the longest of all the five workloads), Task-2 and Task-3 could reach up to around 180,000 step checkpoints while Task-4 would struggle to reach a 68,000 step checkpoint, our standardized evaluation checkpoint. This not only delayed progress, but also limited the amount to which Task-2 and Task-3 could be evaluated at for a valid comparison.

Tasks 2-4 Discussion: The results derived from Tasks 2, 3, and 4 were unexpected. A chart comparing the MPKI values of the tasks with LRU and PARROT is shown in Fig. 4. Not all workloads have a corresponding PARROT MPKI values [3].

As can be seen in Fig. 4, Task-2 and Task-3 policies perform near identically on all workloads. Although this could have been predicted due to the two tasks having extremely similar architectures, it was predicted that the additional layer of Task-3, with more learnable parameters, would still produce some kind of improvement. This was not the case and in some cases Task-2 actually outperformed Task-3 in MKPI as well as Normalized Cache Hit Rates. More surprisingly, Task-4, which has a much more complex architecture with more parameters and more cache state information, did not generate a pronounced improvement in performance over Tasks 2 and 3. It does show a slight improvement for the lbm, milc, and omnetpp workloads, but performs near equivalently for astar and lbm and is significantly worse for mcf.

What is the most surprising however, is how Tasks 2-3 performs seemingly as well in MPKI as PARROT. For Task-4, it is feasible to see how the RNN based network could do almost or just as well as the LSTM based network, as their functionality with the hidden state is very similar hold the cell state. However, Tasks 2 and 3 do not receive any cache access history at all and only see cache lines one instance at a time. Moreover, they can only form linear relationships between the cache access and a cache line while the original PARROT model allows the model to see trends of the cache over time and contextualization between accesses and the cache lines.

Multiple hypothesis can be considered for explaining the performance of Tasks 2 and 3:

1) Hypothesis 1: There is an undetected error or incorrect implementation in our construction of their model architectures. This is possible as we are not completely fluent with the *cache_replacement* code base and there could be a necessary change somewhere in the code that was overlooked. However, as the models trained and evaluated without errors and only two code files were changed to achieve the new architectures, we are unsure where this could be [4].

2) Hypothesis 2: In the attention mechanism of the original PARROT model, the mechanism receives the LSTM hidden layers, containing information on the current and past cache accesses, and the embeddings of each cache lines. Hypothesis 2 considers the possibility that the presence of so much information can "bog down" and confuse the model from discerning which relationships and connections really matter. Therefore, in the case of Task-2 and Task-3 where the model only sees the current cache access and as single cache line at a time, it can can create deep connections between the cache access and each line independently. These deeper connections may increase performance, but without the presence of history, it results in only similar performance as PARROT. This hypothesis is unlikely, though, as the purpose of the attention mechanism is for handling this much information at once and forming

[3]Not all workloads have corresponding PARROT MPKI values due to the addition of workloads later in the project. Unfortunately we ran out of time to train PARROT on the missing workloads before submission. This is something that will be added in the final project.

[4]The possibility of a bug in the model architecture is going to continue to be analyzed in the final project by A.J.

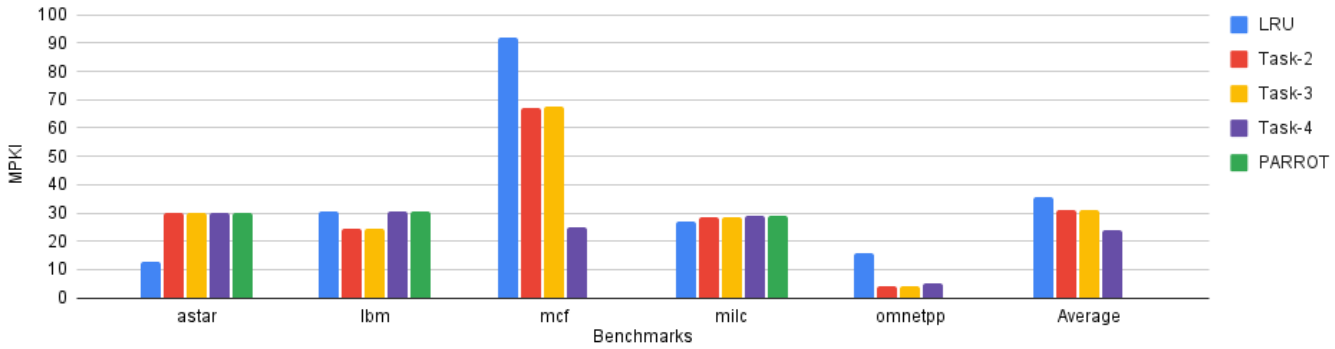

Fig. 4: Single and multi-layer perceptron input format.

relationships between them. Additionally, it is assumed that the original authors tried simpler, linear layer models and improved upon them with PARROT.

3) Hypothesis 3: We believe that another possible reason for this observation, could be the nature of the programs in our benchmarks. It could be possible that the Parrot model overfits on the access patterns. Another possibility is that for these particular benchmarks the history of PCs doesn't carry a lot of information for the model to learn better than MLP, although we believe this is unlikely. In either case, we need to investigate into this further.

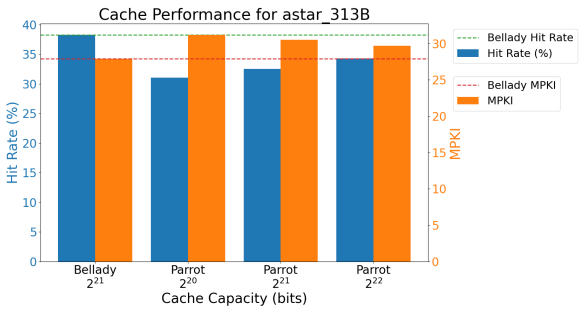

Fig. 5: Table VII visualized

F. Task-5

Objectives: *Task-5* deals with understanding the influence of cache size on the performance of the PARROT replacement policy.

Methodology: We update the cache configuration, changing the `cache_line_size` parameter. Each time this parameter gets updated, a new policy gets trained for every one of our three chosen traces.

Results: Our results for the *astar_313B*, *milc_409B* and *mcf_250B* traces are summarized in VII, IX and VIII, respectively.

Discussion: The values given in tables VII, IX and VIII affirms the general intuition that larger cache sizes lead to increased hit rates, where in the limit, as cache size goes to infinity, hit rate goes to one. This trend can be seen for each of the three test traces.

Challenges: Same as in *Task-6*.

Trace	Cache Capacity	Hit Rate (%)	MPKI	Replacement Policy
mcf_250B	2 MB	46.71	65.96	Belady
mcf_250B	1 MB	44.41	68.82	PARROT
mcf_250B	2 MB	45.52	67.45	PARROT
mcf_250B	4 MB	45.99	66.87	PARROT

TABLE VIII: Cache performance for the *mcf_250B* trace.

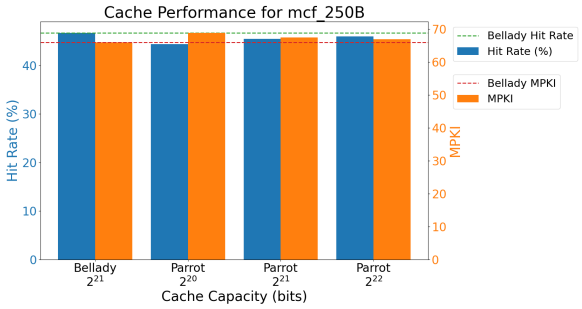

Fig. 6: Table VIII visualized

Trace	Cache Capacity	Hit Rate (%)	MPKI	Replacement Policy
astar_313B	2 MB	38.26	27.90	Belady
astar_313B	1 MB	31.02	31.18	PARROT
astar_313B	2 MB	32.52	30.49	PARROT
astar_313B	4 MB	34.29	29.69	PARROT

TABLE VII: Cache performance for the astar_313B trace.

Trace	Cache Capacity	Hit Rate (%)	MPKI	Replacement Policy
milc_409B	2 MB	6.54	25.19	Belady
milc_409B	1 MB	2.76	26.21	PARROT
milc_409B	2 MB	2.78	26.20	PARROT
milc_409B	4 MB	3.57	25.99	PARROT

TABLE IX: Cache performance for the *milc_409B* trace.

Trace	RNN Type	RNN Activation	RNN Hidden Size	Embedding Type	Embedding Size	Hit Rate (%)	MPKI	Policy
astar_313B	N/A	N/A	N/A	N/A	N/A	38.26	27.90	Belady
astar_313B	GRU	tanh	128	dynamic-vocab	64	31.96	30.75	PARROT1
astar_313B	LSTM	tanh	128	byte	64	28.89	32.14	PARROT2
astar_313B	LSTM	tanh	128	dynamic-vocab	128	32.72	30.41	PARROT3
astar_313B	LSTM	tanh	128	dynamic-vocab	64	32.45	30.53	PARROT4
astar_313B	LSTM	tanh	128	dynamic-vocab	32	32.42	30.54	PARROT5
astar_313B	LSTM	tanh	256	dynamic-vocab	64	32.39	30.56	PARROT6
astar_313B	RNN	relu	128	dynamic-vocab	64	31.89	30.78	PARROT7
astar_313B	RNN	tanh	128	dynamic-vocab	64	32.06	30.71	PARROT8

TABLE X: Hyper-parameter search for the *astar_313B* trace

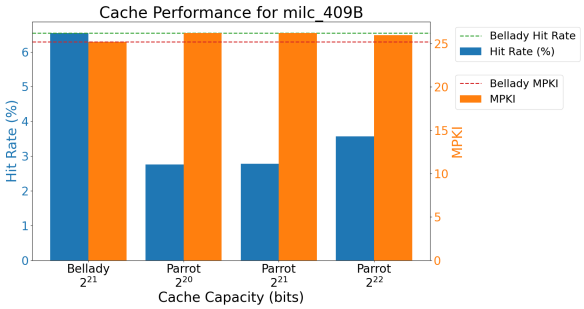

Fig. 7: Table IX visualized

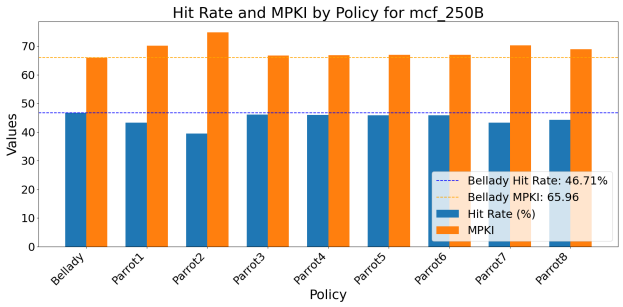

Fig. 9: Table XI visualized

G. Task-6

Objectives: *Task-6* delves into architecture searching with the goal being finding a set of optimal hyper-parameters specifying the cache replacement policy's neural network.

Methodology: We vary several hyperparameters when searching for an optimal replacement policy model architecture, including the type of used recurrent neural network (`RNN`, `GRU`, `LSTM`) the recurrent neural network's internal nonlinearity (`relu` and `tanh`) and the internal hidden vector size (`128`, `256`). Moreover, we vary the embedding type (`byte`, `dynamic-vocab`) and finally embedding size (`32`, `64`, `128`).

Results: The results for the *astar_313B*, *milc_409B* and *mcf_250B* traces are summarized in X, XII and XI, respectively.

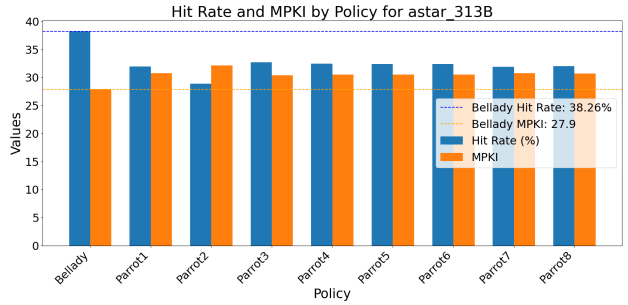

Fig. 8: Table X visualized

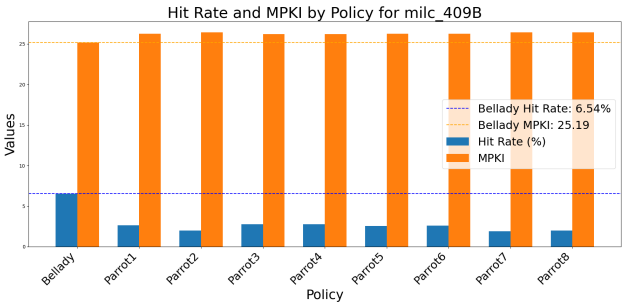

Fig. 10: Table XII visualized

Discussion: Generally, the performance of the replacement policy does not depend on the choice of the recurrent neural network, whether it be `LSTM`, `GRU` or `RNN`. The parameter that plays a major role is the embedding type. Opting for `dynamic-vocab` embeddings typically leads to higher hit rates and correspondingly to lower MPKI values. Both the size of the recurrent neural network's hidden vector as well as the embedding size have an influence on the final performance. Both, when increased, yield policies with higher hit rates.

Challenges: We put a lot of effort into making the training code as scalable as possible, allowing us to submit hundreds or even thousands of jobs on the HPC cluster. While useful, this proved to be a challenging task as debugging, storing and evaluating all the necessary files is difficult, especially when working in a remote environment.

Despite that, we believe the efforts were worth-while as in the future we could perform more architecture-space searches with ease and little effort, utilizing hundreds or even thousands of cluster nodes.

Future Work: Plot Attention Layer as shown in Glider [14],

Trace	RNN Type	RNN Activation	RNN Hidden Size	Embedding Type	Embedding Size	Hit Rate (%)	MPKI	Policy
mcf_250B	N/A	N/A	N/A	N/A	N/A	46.71	65.96	Belady
mcf_250B	GRU	tanh	128	dynamic-vocab	64	43.27	70.24	PARROT1
mcf_250B	LSTM	tanh	128	byte	64	39.52	74.88	PARROT2
mcf_250B	LSTM	tanh	128	dynamic-vocab	128	46.10	66.73	PARROT3
mcf_250B	LSTM	tanh	128	dynamic-vocab	64	46.02	66.84	PARROT4
mcf_250B	LSTM	tanh	128	dynamic-vocab	32	45.89	66.99	PARROT5
mcf_250B	LSTM	tanh	256	dynamic-vocab	64	45.90	66.98	PARROT6
mcf_250B	RNN	relu	128	dynamic-vocab	64	43.26	70.25	PARROT7
mcf_250B	RNN	tanh	128	dynamic-vocab	64	44.32	68.94	PARROT8

TABLE XI: Hyper-parameter search for the *mcf_250B* trace

Trace	RNN Type	RNN Activation	RNN Hidden Size	Embedding Type	Embedding Size	Hit Rate (%)	MPKI	Policy
milc_409B	N/A	N/A	N/A	N/A	N/A	6.54	25.19	Belady
milc_409B	GRU	tanh	128	dynamic-vocab	64	2.65	26.24	PARROT1
milc_409B	LSTM	tanh	128	byte	64	1.98	26.42	PARROT2
milc_409B	LSTM	tanh	128	dynamic-vocab	128	2.78	26.21	PARROT3
milc_409B	LSTM	tanh	128	dynamic-vocab	64	2.75	26.22	PARROT4
milc_409B	LSTM	tanh	128	dynamic-vocab	32	2.54	26.27	PARROT5
milc_409B	LSTM	tanh	256	dynamic-vocab	64	2.58	26.26	PARROT6
milc_409B	RNN	relu	128	dynamic-vocab	64	1.92	26.44	PARROT7
milc_409B	RNN	tanh	128	dynamic-vocab	64	2.00	26.42	PARROT8

TABLE XII: Hyper-parameter search for the *milc_409B* trace

to get deeper insights.

H. Task-7

Objective: To compare our experimented neural models with the existing cache replacement policies.

Methodology: ChampSim has provided the implementation for LRU, SHiP, SRRIP and DRRIP replacement policies. We simulated SHiP and SRRIP policies on ChampSim to get the LLC statistics and calculate the MPKI, miss rate and hit rate values for comparison with our neural models.

- **SHiP:** Signature-based Hit Predictor (SHiP) improves the cache replacement decisions by predicting the reuse of cache lines. The approach calculates a signature from CPU features like program counter, instruction sequence history, memory address and memory access behavior. This is a learning-based solution implemented using tables and counters, which gives an outcome for re-reference used to predict reuse of a cache line.
- **SRRIP:** Static Re-reference Interval Prediction (SRRIP) is a replacement policy that predicts re-reference interval to make the eviction decisions. The solution uses Re-reference interval Prediction Value (RRPV) for every cache line to make make eviction decision. On cache insertion, the cache line receives an intermediate RRPV value which increases due to inactivity and is reset to 0 on cache hit. Therefore, the line with highest RRPV value is evicted.

It is key to understand the crucial terminology difference between the two policies. Although the fundamental concepts in both the policies are similar, there is a subtle difference between the solutions proposed by them. Reuse is an probabilistic indication of whether a cache line is likely to be used in the future or not (**if?**), whereas re-reference interval captures the temporal likeliness future access(**when?**).

Results: The results for this task are shown in Figure 11 and in the Appendix.

Discussions: Learning-based approaches like SHiP and SRRIP improve on heuristic-based solutions like LRU. Neural or Machine Learning-based approaches aim to improve on these solutions. Therefore, SHiP and SRRIP are good choices for comparison.

As seen in Figure ,

- **mcf_250B:** Learning-based SHiP and SRRIP policies perform better than LRU. Perceptron-based policies perform even better and RNN outperforms all by performing more than 3 times better than LRU.
- **astar_313B:** SHip and SRRIP perform slightly worse than LRU but MLP and RNN perform approximately two times better than all of them.
- **lbm_564B:** LRU, SHiP and RNN perform equally well while SRRIP performs slightly worse than them. Perceptron-based policies perform the best
- **milc_409B:** All the policies perform almost equally well
- **omnetpp_4B:** SHiP and SRRIP policies perform better than LRU. Perceptron-based policies outperform everyone and while RNN performs slightly worse than them.

A look at the average MPKI values over these benchmarks shows a general trend of improvement from heuristic solutions to learning and a significant improvement as we go to ML-based solutions. Therefore, based on the results seen, we can conclude that ML-based neural policies are promising for the Cache Replacement Problem.

I. Task-8

Objective: This task aims to perform a literature review on feature engineering and performing experiments on different input features

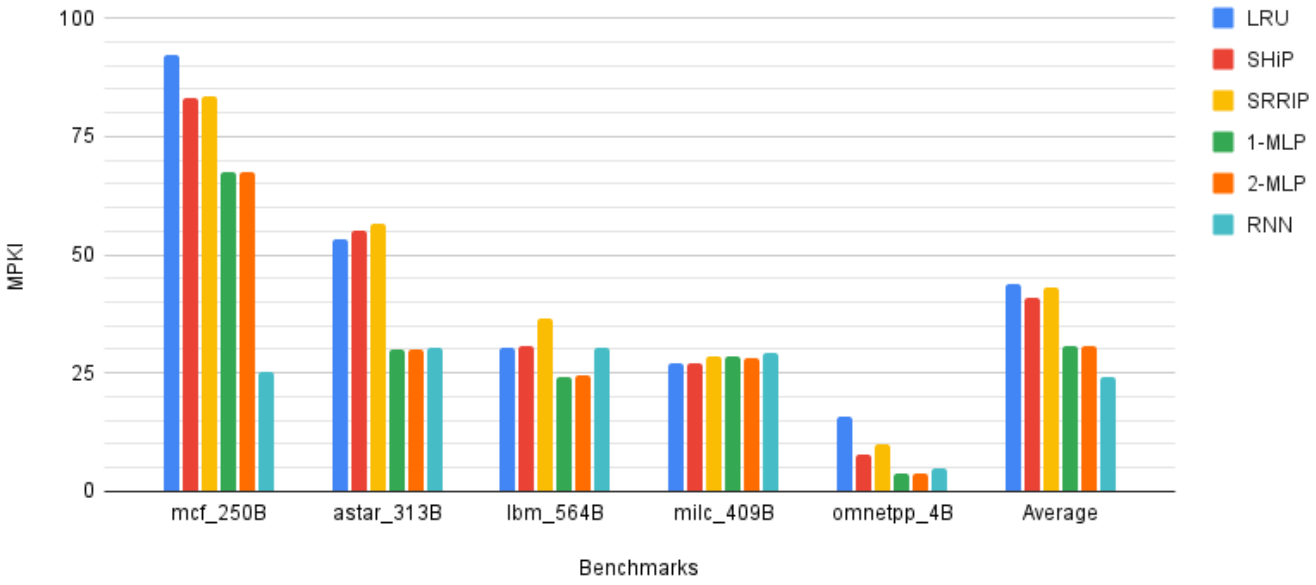

Fig. 11: MPKI results for different cache replacement policies

Paper	Features	Application	Model Architecture
BLBP [4]	Indirect branch address	Branch Prediction	Perceptron
PBNP [6]	PC	Branch Prediction	Path-based Neural Branch Prediction algorithm
GHRP [11]	PC, control-flow history	Reuse Prediction	Hash-based Prediction algorithm
MPPPB [7]	PC, physical memory address, bias, burst, insert, last miss, offset	Reuse Prediction	Perceptron
ChiRP [12]	Global path history, conditional branch history, unconditional branch history, PC	Predictive TLB Replacement Policy	Hashing-based Algorithm
Hermes [2]	PC, load PC, virtual address, virtual page number, first access, cache-line offset	Off-chip load prediction	Single Layer Perceptron
DBRB [8]	Block signature	Dead Block Prediction	Hash-based Predictors
PPF [3]	Physical address, Cache line, Page Address, PC, Confidence	Increase Prefetcher Coverage	Perceptron
PerSpectron [3]	A subset of highly correlated CPU features	Identification & Classification of Attacks	Perceptron
EVAX [1]	A subset of highly correlated CPU features	Defense against microarchitectual attacks	GAN-based solution
RLR [13]	access preuse, line preuse, line last access type, line hits since insertion, and line recency	Cache Replacement Policy	RL based online algorithm

TABLE XIII: Table of Features used in Previous Work

Methodology: TableXIII summarizes the input features used in related works on ML based solutions to microarchitecture problems.

The authors have chosen a variety of relevant CPU parameters depending on the application. They then use combinations of relevant features by performing operations like - XOR, hash indexing, concatenation and bits extraction. Based on the results of experimentation, heat maps of input features are used to determine the best and most useful input features.

A key observation made during literature review is that authors tend to use features generated by performing XOR operation on CPU components like PC, physical address, virtual address, branch history, etc. We believe that this tends to capture the essence of both the components in one feature without significant loss of information (like in AND or OR operations). Therefore, we can effectively give similar amounts of information using one feature.

With this insight we decided to modify the existing input features used in the PARROT infrastructure by using (PC$\oplus$memory address, memory address) in place of (PC, memory address) as our input features. We chose to replace PC and not memory address because memory address is used for accessing cache sets. Giving a modified input feature as memory address would result in a different caching behavior. Thus, we were able to test out a different input feature without making changes to the existing PARROT model architecture or code base.

We wrote a python script to create new trace files (train, test, valid) with the newly featured input using the older trace files. This ensured testability across different model architectures

without making changes in their respective codes.

Results: Due to failure of HPC jobs, with an ECC hardware error, we could train but not evaluate this method in time.

Future Work: Extract more parameters from ChampSim apart from the PC and memory address to create and experiment with different input features.

J. Task-9

Objective: Analyze the behavior of attention layers and compare with previous research.

Let's specifically discuss the attention layers we were able to generate out of the randomly initialized model versus the astar workload. The following are the results of the attention layer these two models outputted.

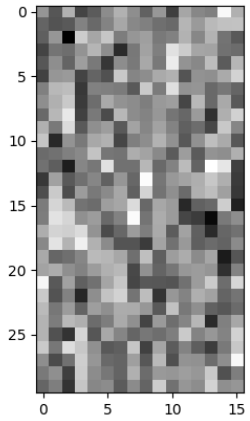

Fig. 12: Attention layer from a randomly initialized model generated on the *astar_313B* trace

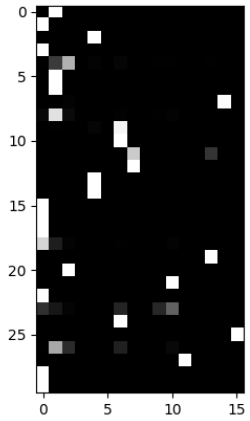

Fig. 13: Attention layer from a trained model generated on the *astar_313B* trace

The y-axis of the layer represents the timestamps as the model is being applied. On the other hand, the x-axis represents the cache lines of the model and how much "attention" and relevance each cache line holds during a specific timestamp value. This relevancy is measured for each of these cache lines based on the intensity value of the coordinates on the layer. The lighter the coordinate pixel is, the more relevant and more attention is being given. White pixels represent the most attention and relevancy and black pixels represent the least attention and irrelevancy to that cache line in that specific timestamp. Now looking at the attention layer of the randomly iterated model, it is extremely difficult to identify a pattern and what is going on in the algorithm. The attention appears to be random, giving various amounts of attention to random cache lines as time progresses. Once we reach the end of the time frame the model is running, it is practically impossible to come to a conclusion to what occurred and was computed throughout the attention layer. In contrast, let us acknowledge the resulting attention layer of the astar workload. Comparing this to the randomly iterated model's attention layer, it is easier to identify what is happening to certain cache lines of each timestamp in the attention layer. To reiterate, astar is a path finding algorithm that finds the number of existing paths and their traveling distances in a environmental map in computer games. When we apply this information to the attention layer with the path-finding algorithm in mind, we can theorize that the workload is searching for the most efficient paths of the inputted map and finds those paths and validates the accuracy of the results at the end of the workload. It can also be observed that there is a certain amount of attention that can be given at every time stamp and if multiple cache lines are relevant, this attention is distributed based on how relevant they are in that point in time. If only one cache line is relevant, it has all the attention from the workload at that timestamp and therefore its pixel is completely white. With this observation, we have the idea that as time progresses, certain cache lines become relevant and are paid attention to as they assist in the path-finding algorithm. We can also infer that once sub-algorithms are complete at specific cache lines at certain points of time, they become less relevant in later timestamps as their computations are complete and are already applied to the bigger picture of astar's overall algorithms. Some of these cache lines have more minimal attention compared to others and this may be due to the weight of their contributions to the overall workload. If the model or workload has a set goal provided, attention layers can be helpful in analysis of what is happening in certain aspects at different points of time. The information gathered from here can help better understand how the workload functions and how its algorithms takes in its inputs to get the potentially desired output.

K. Task-10

Objective: Understand the behavior and dependencies of SPEC CPU workloads.

1) Overview of Workloads: Let us first go over the high-level semantics of SPEC CPU workloads. The following are descriptions of some of these workloads.

2) lbm: The LBM benchmark is written in ANSI C. It implements LBM also known as "Lattice Boltzmann Method" and is meant to be used in the material science field. It

simulates incompressible fluids and their behaviors in a free surface in 3D, particularly the formation and movement of gas bubbles in metal fluids and foams. For the sake of testing and benchmarking and optimization for various architectures, the case maximizes its use of macros that limit data access.

The input of the LBM program calls for several command line arguments such as <time steps>, <result file>, <0: nil, 1: cmp, 2: str>, <0: ldc, 1: channel flow> and [<obstacle file>]. <time steps> is the number of time steps that should be executed before storing the output. <result file> simply contains the name of the file. <0: nil, 1: cmp, 2: str> tells the program what specific action should be done to the result file given. If the argument is zero, it does nothing; when it is one, the computed results are compared to the results that are stored in the result file; and when it is two, it stores and overwrites the computed results into the result file. <0: ldc, 1: channel flow> chooses between two simulation setups where argument of zero sets up lid-driven cavity (LDC) where "shear flow is driven by a 'sliding wall' boundary condition" and argument of one steps up the simulation to "flow driven by inflow/outflow boundary conditions". Lastly, the <obstacle file> is an optional argument that loads an obstacle file as an additional input of the free surface. The output of this benchmark program is various depending on the action argument given in <0: nil, 1: cmp, 2: str>. When action 2 was requested, the result file that contains the 3D velocity vector for each cell is stored. The default file format of this output is a sequence of binary single precision values with a specific ordering. This default file format can't be altered from the command line but the output precision can be changed to double precision through `config.h`. If the action taken is set to one, the program instead returns the maximum absolute difference of velocity after comparing each cell in the result file individually.

3) astar: The purpose of the astar benchmark is to find pathways in a map for a computer game based on the map's terrain, and the movement speed. This benchmark uses C++ and accepts inputs that may include maps formatted in binary and forest-oriented test maps. With this input, it can apply three of the path-finding algorithms that are a part of the astar library. One applies the A* algorithm to find passable and non-passable terrain types in the map. Another algorithm modifies the first algorithm and factors in different terrain types and movement speeds. Lastly, the third algorithm applies the modified A* for graphs which is created based on the inputted map regions. Additionally for map region determination, this library includes pseudo-intellectual functions available for use. The output from applying this program is the number of existing ways to simulate the program and the total length needed to validate correctness. As of right now, there are no known portability issues with this library.

4) milc: The MILC benchmark contains a set of code in the C language developed by MIMC Lattice Computation (MILC). The purpose of this benchmark is to perform simulations of four dimensional SU(3) lattice gauge theory on MIMD parallel machines. The input applied into this benchmark is an input file that contains data sets, test, train and ref as well as their various grid sizes. The parameters contained in the input file contain the dimensions of the grid, qualities of quarks such as quantity, and mass, trajectories to setup and run measurements,and how many steps needed for simulations and trajectories etc.. The output of the benchmark is used to verify correctness. Additionally, the original code can possibly be compiled to provide helpful timing information using portability flags such as -DCGTIME -DGFTIME -DFFTIME. For validation purposes, these flags are set to low and turned off in SPEC CPU 2006.

5) leslie3d: The LESlie3d benchmark, programmed in Fortran 90, is derived from "Large-Eddy Simulations with Linear-Eddy Model in 3D" which is a researching-level computational fluid dynamics program. Its primary purpose is to "investigate a wide array of turbulence phenomena such as mixing combustion, acoustics and general fluid mechanics". In the specific case of CPU 2006, this program is set up in a way to resolve test cases that factor in these turbulence phenomena to better understand turbulent physics. This program has three various input stack sizes, test, train and ref available for test cases. On the other hand, the input parameters that interact with these variables include grid size, flow parameters and boundary conditions. In contrast, the output that comes from applying this benchmark is a text file that contains information based on analysis from the simulation and tracks momentum thickness overtime.

6) GemsFDTD: The GemsFDTD benchmark is a subset program in the Fortran 90 language that GemsTD developed in the General Electromagnetic Solvers (GEMS) project. Its purpose is that it "solves Maxwell equations in 3D in the time domain using the finite- difference time-domain (FDTD) method". This benchmark has three steps: initialization, time-stepping and post-processing, where time-stepping takes up almost all of the programming run time. The inputs placed into this benchmark includes a main input file that needs to have the specific name of "yee.dat" a PEC description file, and primary keywords alongside several secondary keywords could be given if necessary. The primary keywords are used by the benchmark to define the problem size, number of time steps needed, cell size and the CFL value. It should also be noted that the order in which the primary and secondary keywords are given in yee.dat doesn't have to follow a specific ordering. These inputs should give the outputting result of an ASCII file that contained the RCS data that was requested of the program. This file should be in a .nft file named based on the input file that contains the primary keyword of "NFTRANS_TD" (must be all capital letters) and the secondary keyword "Filename-base". The RCS of the PEC object can be plotted using Matlab scripts such as farfieldgemsTD.m and rcsmain.m. These scripts should be included in the benchmark.

7) omnetpp: The omnetpp benchmark is programmed in C++ that creates and applies a simulation of a large Ethernet network. The simulation this benchmark recreates is based off of an open source simulation framework from OMNeT++. OMNeT++'s architecture is flexible enough that it can be generalized for scenarios outside of simulation of communication such as simulations of IT systems, queuing networks, hardware architectures and business processes as well. The input parameters for the benchmark are specified in a omnetpp.ini file. The parameters found in this input file

include the number of switches on the backbone, number of LANs and hosts of various sizes on each backbone switch, host configuration and parameters for setting up the traffic model. Running this benchmark generates several, various statistics. Nodes can be modified that allow them to record more statistics beyond basic information such as number of frames sent, received, dropped, etc..

L. Task-11

Objective: The objective of Task-11 is to generate a RAG file encompassing a mass amount of metrics and results generated from previous tasks to provide to an LLM for interpretation.

Methodology: To generate a RAG file that incorporates a large amount of model performance data, a custom script, raggen.py, was created. It takes the essential code blocks from cache/main.py to simulate cache eviction policies and log their decisions. Specifically, this code takes checkpoints from the policies trained in Tasks 2-4 as well as a LRU and Beladys policy and simulates them across all 5 workloads for a variable number of cache accesses. The current workload, current cache access PC, policy name, and PC of the evicted cache line are sequentially logged.

Results: A sample RAG file with 100 cache access for each workload can be seen here.

Challenges: The biggest challenge in this task was getting familiar with the `langchain` library, making sure we are using the all its bits and pieces correctly and feeding them the right information. Studying `langchain`'s documentation required a considerable amount of time, especially due to the library's ever-present levels of abstraction.

The creation of the custom `raggen.py` code made it extremely simple to piece together essential parts of the code to achieve the desired functionality without any of the necessary logging or initializations. Similar to Legos. (Props to Jakub).

M. Task-12

Objective: *Task-12* goes beyond using a general purpose large language model for question answering by employing the RAG file generated in subsection III-L. Our goal was to create a simple LLM application with an easy-to-use `python3` interface for getting insights about cache replacement policies, traces and other cache eviction tasks.

Methodology: To create our LLM application, we followed the LangChain RAG App, a popular python library specializing in building LLM-based software. The high-level process of our RAG is as follows:

1) Generate a text file containing large amount of eviction data from multiple policies. (Task-11)
2) Split the text file into 1,000 character chunks of text and embed each text chunk with the OpenAI textual embedder.
3) Generate and prompt an LLM with cache replacement questions.
4) Upon input, the questions will themselves be embedded and compared with the text chunk embeddings using cosine similarity. The text chunk with the highest similarity is then chosen as the relevant data and the information of that chunk is then retreived from the text file and stored as context for the input question.

The interface for our application is simple, with the only two parameters being the path to the previously generated RAG file and a query (question) to be asked. Our system then processes that question, retrieves the relevant pieces from the RAG file, passes that information to the underlying LLM and finally relays the LLM's output back to the user.

Results: In this task, we only employ *zero-shot* prompting. More advanced prompting techniques are employed and tested later in this paper in subsection III-N. Below, we list a set of questions of increasing difficulty and the corresponding answers to those queries. More examples can be found in our repository[5]. In *Task-12*, we were using the `gpt-3.5-turbo` model from OpenAI.

Discussion:

Generally, our Q&A application works well most of the easier questions and even for some more difficult to answer queries where the relevant pieces of information are scattered throughout the RAG file.

Challenges:

Getting the simplest of Q&A application up and running, thanks to the numerous examples on `langchain`'s website, proved easy, however getting all the details right, making sure we are sending `POST` requests to OpenAI as little as possible did require quite a bit of effort. This has been solved by introducing RAG vector database caching.

N. Task-13

Objectives: *Task-13* extends *Task-12* by incorporating more advanced prompting techniques (`one-shot prompting`, `self-consistency prompting`). Moreover, this task utilizes an LLM as an eviction policy. Finally, as part of *Task-13*, we fine-tuned the `meta-llama/Llama-3.2-1B` model[6] for cache eviction task.

Methodology: We implemented `one-shot` prompting by including a single input-output example pair, added to the original `zero-shot` prompt used in *Task-12*.

The implementation of `self-consistency` prompting is done by means of prompting our model N times, storing the models responses in a list and finally asking the model one more time with the N previous responses included, instructing it to generate one last response. For this to work, the model's `temperature` parameter has to be set to a value greater than zero.

We used the `gpt-3.5-turbo` model from OpenAI as an LLM of choice for our experiments with the LLM behaving as an eviction policy. We are instructing the model to choose a cache line to evict. The model is instructed to choose a cache line to evict. Behind the scenes, the model gets access to all the data generated in *Task-11* in our RAG file.

[5]https://github.com/lnotspotl/raggen/tree/main/task12
[6]https://huggingface.co/meta-llama/Llama-3.2-1B

Finally, we fine-tuned the `meta-llama/Llama-3.2-1B` model, trying to make it immitate the Belady's policy. The model is given the current `PC` as well as the current `address` together with information about the present `cache_lines`. With that information, the model is fine-tuned to output a single number representing the number of the cache line to be evicted. While fine-tuning gives us more confidence that the generated output will contain valid information, in case the output is invalid (e.g. cache line number not present or the output cannot be parsed), we fall back to a random eviction policy. Most of the times, however, our fine-tuned model produces valid outputs.

Discussion: `zero-shot` prompting works good most of the times, however there are cases where the question is answered incorrectly. One such example is question 8[7], where the model incorrectly identifies just two out of three pieces of information we are searching for. The remedy for this issue turned out to be `self-consistency` prompting. When using `self-consistency` prompting, our Q&A LLM was able to answer question 8 correctly.

Method	Number of Close-ended Questions Answered Correctly
Zero-Shot	8/10
One-Shot	8/10
Self-Consistency	10/10

TABLE XIV: Comparison of Correct Answers Across Prompting Techniques

Question No.	Question
1	What is cache memory, cache set, PC?
2	What is a cache hit and a cache miss?
3	Could you please look at the memory trace of astar benchmark from accesses 1 to 100 and find an example of eviction where policy matches belady?
4	Could you please look at the memory trace of astar benchmark from accesses 1 to 200 and find an example of eviction where policy matches belady?
5	What did policy02 evict when the PC was 0x401e1b
6	What is the structure of the context
7	What eviction policies are being tested and what policy is the reference optimal policy?
8	What traces are included?
9	Generally, would you expect policy02 to outperform policy04? Talk about each of the traces files individually
10	What traces are included? (with summary)

TABLE XV: List of 10 Questions We used for LLM testing

We even tried `one-shot` prompting, however finding the right example question-answer pairs proved to be cumbersome and thus we did not invest much time into that.

Using a general-purpouse largue language model as an eviction policy turned out to be futile. No matter our prompts and other information retrival techniques used, we were not able to make our LLM work good enough to justify it's usage as a cache replacement policy. In fact, it perform worse than a random eviction policy. Morever, inference times are orderes of magnitude higher for the LLM, rendering it's use in real-life applications impossible.

Challenges:

[7]https://github.com/lnotspotl/raggen/tree/main/task12

While we were able to fine-tune a proof-of-concept `Llama`-based eviction policy on a simplified trace, fine-tuned `LLama` models specifically trained for each trace are still to be created. Training these models proved to be more time-consuming than we had initially expected. As of writing this report, six different `Llama` models (3x one billion parameter variant, 3x eight billion parameter variant) are being trained on the `mcf_250B`, the `milc_409B` and finally the `astar_313B` traces, larger and smaller `Llama` model variant for each trace. Writing the actual training code also proved to be challenging as the training itself necessitates training in 8-bit precision. Due to that, we had to learn how to use several python libraries, including `peft`, `accelerate`, `qlora` and others.

Due to time constraints, we were unable to properly explore the high-level semantics for traces.

REFERENCES

[1] S. M. Ajorpaz, D. Moghimi, J. N. Collins, G. Pokam, N. Abu-Ghazaleh, and D. Tullsen, "Evax: Towards a practical, pro-active adaptive architecture for high performance security," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1218–1236.

[2] R. Bera, K. Kanellopoulos, S. Balachandran, D. Novo, A. Olgun, M. Sadrosadat, and O. Mutlu, "Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1–18.

[3] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 1–13.

[4] E. Garza, S. Mirbagher-Ajorpaz, T. A. Khan, and D. A. Jiménez, "Bit-level perceptron prediction for indirect branches," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 27–38.

[5] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 78–89, 06 2016.

[6] D. Jimenez, "Fast path-based neural branch prediction," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 243–252.

[7] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 436–448. [Online]. Available: https://doi.org/10.1145/3123939.3123942

[8] S. M. Khan, Y. Tian, and D. A. Jiménez, "Sampling dead block prediction for last-level caches," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 175–186.

[9] P. Lewis, E. Perez, A. Piktus†, F. Petroni†, V. Karpukhin†, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2020. [Online]. Available: https://arxiv.org/abs/2005.11401

[10] E. Z. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," 2020. [Online]. Available: https://arxiv.org/abs/2006.16239

[11] S. Mirbagher Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez, "Exploring predictive replacement policies for instruction cache and branch target buffer," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 519–532.

[12] S. Mirbagher-Ajorpaz, E. Garza, G. Pokam, and D. A. Jiménez, "Chirp: Control-flow history reuse prediction," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 131–145.

[13] S. Sethumurugan, J. Yin, and J. Sartori, "Designing a cost-effective cache replacement policy using machine learning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 291–303.

[14] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 413–425. [Online]. Available: https://doi.org/10.1145/3352460.3358319

APPENDIX

CONTRIBUTIONS

A.J. Beckmann

I worked on the following tasks during this project:

- Implemented, trained, and evaluated Tasks 2, 3, and 4 and analyzed their results.
- Modified and used the raggen.py code created by Jakub to generate a RAG file with each policy and each workload for Task-11
- Wrote the introduction, background, Task 2-4, and Task-11 report sections.
- Helped contribute ideas for the project, results, and report.
- Proof read the report.
- Discord activity - 254 messages

Jakub Jon

I worked on the following tasks during this project:

- Helped debug trace generation in task 0 and MPKI calculation in task 1
- Implemented tasks 5, 6, 12, 13
- Collaborated with AJ on task 11
- Written sections for task 5, 6, 13 and partially 11, 12, in collaboration with AJ.
- Written code for attention plot retrieval: for task 9.
- Helped contribute ideas to other tasks
- Proof read the report.
- Discord activity - 342 messages

Anjolie Baccay

I worked on the following tasks during this project:

- Completed discussion and research of workloads and attention layers for Task 9 and Task 10.
- Collaborated with teammates to write the final report.
- Proof read the report.
- Discord activity - 42 messages

Kaushal Mhapsekar

I worked on the following tasks during this project:

- Primarily worked on the Microarchitecture tasks of the Homework.
- Contributed with others to share ideas and assisted teammates in their tasks, when needed.
- Worked with ChampSim and performed investigation for tasks 0 & 1.
- Performed simulations on ChampSim for Task 7.
- Prepared traces with modified input features and performed training & evaluation for task 8.
- Collaborated with teammates to write sections on tasks 0,1,7 & 8 the final report. Also worked on overall formatting and organization of report and spreadsheet.
- Proof read the report.
- Discord activity - 339 messages

We all believe that we have contributed equally towards the homework in our own capacity and skill sets.

MEETINGS:

Our meeting logs are shown below: Many more informal meetings took place after class, in the library, and over Discord, where a lot of the progress and problem solving took place. In these, all team members were active and involved and contributed to the progress of the project.

Date	Time	A J	Anjolie	Jakub	Kaushal
10/09	3 hours	Present	Present	Present	Present
10/18	2 hours	Present	Present	Present	Present
10/24	2 hours	Present	Present	Present	Present
11/04	3 hours	Present	Present	Present	Present
11/11	10 hours	Present	Present	Present	Present

TABLE XVI: Meeting Log

RESOURCES

You can find the links to our works here:

- GitHub - https://github.com/lnotspotl/raggen
- OverLeaf - https://www.overleaf.com/read/scrkcxzsntxd5ef26b
- Results Spreadhseet - Google Sheet

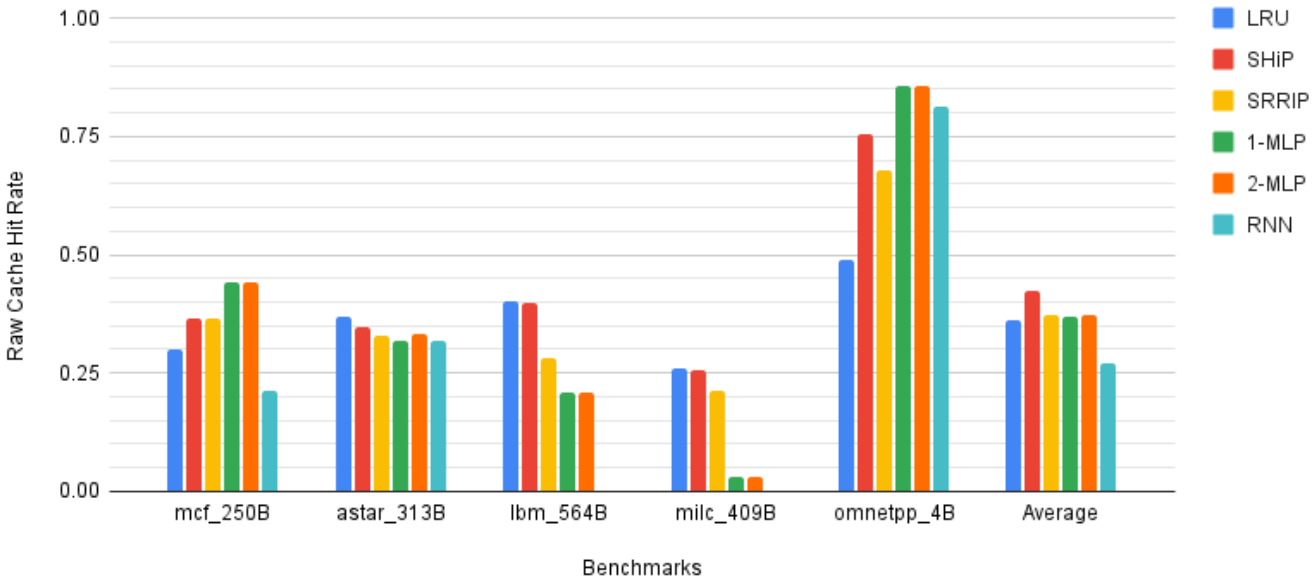

Fig. 14: Raw Cache Hit Rate results for different cache replacement policies (Task 7)

Other Results

We have plotted one of the extra results in Fig. 14. We have included more such interesting results and graphs apart from the ones presented on our GitHub and Google Spreadsheet.