C3PU: An LLM-based CPU Design Assistant

Kaushal Mhapsekar, AJ Beckmann, Jakub Jon, Nishant Raman, Avit Rane,
Naveen Gopalakrishnan , Atharv Oak, Dezmun Roper-Bryant , Anjolie Baccay
Department of Electrical and Computer Engineering, NC State University

***Abstract*—This paper is the third continuation of ECE592-106 Homeworks 1 & 2. Here, we use and expand on our results from Homework 2 to dive into exploration and exploitation of the PARROT model. We explore how feature engineering can improve the performance of a cache eviction policy and we exploit the PARROT model by building a RAG & LLM-based AI chat assistant with the goal of gaining insights to improve the state-of-the-art cache replacement policy. We find that using the memory address tags, rather than the full memory address, is a computationally less expensive feature that can improve the PARROT's cache hit rates on the ASTAR benchmark. Additionally, our AI Assistant is able to comprehend the cache access data from the RAG system and produce meaningful answers with 80% success rate, while also producing python scripts for data analysis. We propose a proof-of-concept for CPU Design AI Assistant in this paper.**

***Index Terms*—Cache Replacement, Attention, Feature Engineering, Large Language Models, RAG.**

I. Introduction

The cache replacement problem is one of the most fundamentally crucial CPU performance problems. When the cache sets are filled, the eviction decision becomes very crucial. In recent years, research for cache replacement problem shifted focus from heuristic-based solutions to learning-based solutions and now onto machine-learning-based solutions.

Machine-learning based solutions encounter two main problems - 1) time, i.e, the time required to infer a prediction or outcome and 2) space, i.e, the storage required to store a highly accurate model. Perceptron-based replacement models gained traction in various computer architecture applications due to their ability to optimize higher degree functions and ease of implementation at the hardware level. These models can capture the nature of cache access patterns and branches, typically better than heuristic solutions and other learning-based solutions.

Glider [6] proposed to train an offline LSTM model that outperformed the then state-of-the-art cache replacement policy which takes advantage of the LSTM's ability to learn over a history of cache accesses and improves performance over perceptron models. The authors then plot attention layer of the model to gain insights from the trained offline model to develop an online model that could match the performance of the offline model, while being hardware friendly. This is a promising methodology wherein we can train offline models that outperform the state-of-the-art and then derive insights from them to implement an online solution on hardware.

This sprouts the need for an LLM application that could have the cache access data and optimal caching decisions which could learn and understand the semantics from high-level program to the level processor and cache accesses. Such an application could then provide deeper insights into selection of features, cache access patterns and evaluate replacement policies.

Hawkeye [2] and Glider [6] were the first to explore learning-based solutions to the cache replacement problem. Both approaches attempt to improve cache performance by predicting the reuse of cache lines. Although they promise a significant improvement in hit rate and speedup (IPC), 8.9% speedup for Glider and a 14.7% speedup for Hawkeye, they have some limitations. Namely, a counter based approach, PC based eviction decisions and limited model space exploration. The Attention mechanism used in Glider helps in getting some wonderful insights. However, it is difficult for an engineer to correlate these plots with high-level C code implementations for all the instructions and benchmarks.

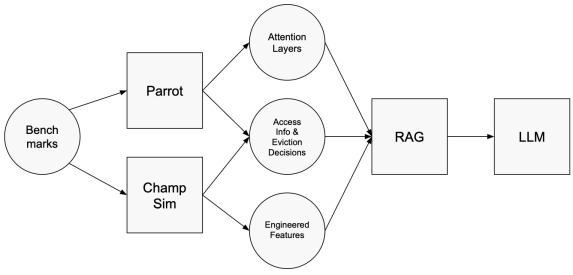

Fig. 1: Overview of our Project Pipeline

Fig. 1 shows the broad overview of the pipeline we followed to build our agent. We used the SPEC CPU 2006 benchmarks in the form of CRC2 championship traces and C source code. These were then passed into ChampSim and Parrot simulators to get Attention Plots, Cache Access information, Cache Eviction decisions,

High-level program semantics and Engineered Input Features. The purpose of these steps is to generate as much relevant data as possible that could be fed to the RAG to aid it in providing good context. The RAG would then use its optimization techniques and provide context to the LLM so that it can provide useful insights.

This work focuses on building a RAG & LLM-based AI chat assistant with the goal of assisting engineers in gaining insights to improve the state-of-the-art cache replacement policy. We achieved a success rate of 80% on the benchmark questions for the specific application of cache replacement.

Our work makes the following contributions:

- Developing a framework to extract and engineer input features from the ChampSim simulator and evaluate their performance for cache replacement.
- Extending the analysis of Attention Layers to multiple SPEC CPU 2006 benchmarks
- Developing an open-source LLM assitant, named C3PU, for the purpose of aiding CPU Design, particularly for the cache replacement problem.

II. Background and Related Works

A. Caches and Replacement Policies

Efficient memory access is critical for maintaining high computational performance, of those key factors is cache management. Particularly how data is replaced within the cache. Caches are small data storages that provide a fast access to frequently used data. However, when it becomes full, data must be evicted to make room for new data. This decision is made by the cache replacement policy.

The Markov Decision Process contains several key components which include the state space, action space, transition dynamic, and reward function. The state space contains three different elements: the cache state, current access, and the access history. The cache state contains the current contents of the caches which have a memory address stored in each one. The current access includes both the memory address being accessed and the program counter with the instruction. The access history captures the patterns that help the model predict future access behaviors. These elements ensure that the cache replacement policy has access to all relevant information to make an informed decision.

The action space depends on whether the current memory access result is a hit or a miss. A cache hit occurs when the data being requested is already stored in the cache, which allows for fast retrieval. A cache miss, on the other hand, happens when the data requested is not found in the cache (Figure 1). Therefore, an eviction must be made to store the information into the cache. The action space consists of deciding which of the lines in the cache set should be evicted to make space for the new memory address in the case of a cache miss. However, if the result is a cache hit, no eviction is necessary, and no action is taken. Therefore, an eviction is only necessary in the event of a cache miss.

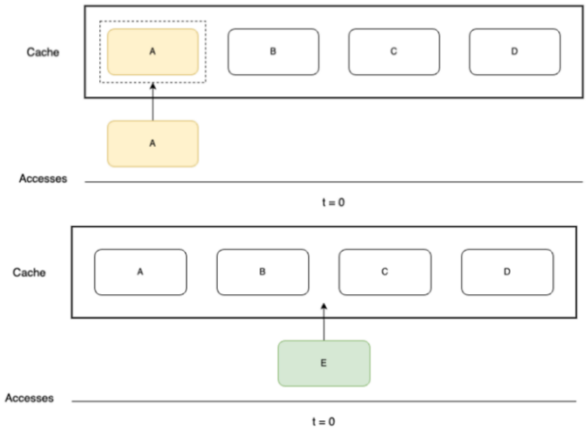

Fig. 2: Cache hit and miss. In the top image at t=0, line A is accessed and is already in the cache, causing a cache hit. In the bottom image at t=0, line E is accessed and is not in the cache, causing a cache miss.

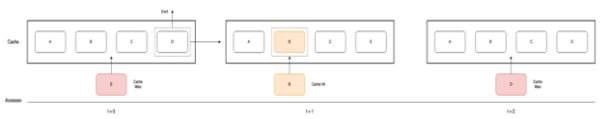

Fig. 3: Cache transition and evolution. As a result of a cache miss, the cache will evict a line and insert a new one. Therefore, as t increases, the state of the cache will change.

The cache state will change in response to the memory access and changes made in the cache. When memory access results in a cache hit, the state of cache will not change. But in the result of a cache miss, the policy must evict one of the existing cache lines to make room for the new data. The eviction will depend on the cache replacement policy used. Once the policy decides which line to evict the state of the cache is updated, and the new block is inserted into the cache (Figure 2). Therefore, the transition dynamics are driven by the cache hits and misses which evolve over time as it responds to the memory access and policy eviction decisions.

The reward $R(s_t)$ is 0 for a cache miss and 1 otherwise for a cache hit. The goal is to learn a policy $\pi_\theta(a_t \mid s_t)$ that maximizes the total number of cache hits possible for a sequence of different cache sequences

$$(m_1, pc_1), \ldots, (m_T, pc_T).$$

The reward function is essential for the policy to learn to make eviction decisions that reduce the number of cache misses.

The cache replacement problem involves making a series of decisions over time that affect future states and outcomes. Because of the inherently stochastic nature of cache replacements and the fact that the dynamically changing state space is partially controllable, the cache replacement algorithm is formulated as a Markov-Decision Process (MDP). A Markov-Decision Process, or MDP for short, provides engineers with a framework for sequential decision making where outcomes are partially unknown. In this paper, an imitation learning paradigm utilizes an MDP to derive effective cache management strategies. An MDP is characterized using states (S), actions upon those states (A_s), the reward function (R), and the transition dynamics (P).

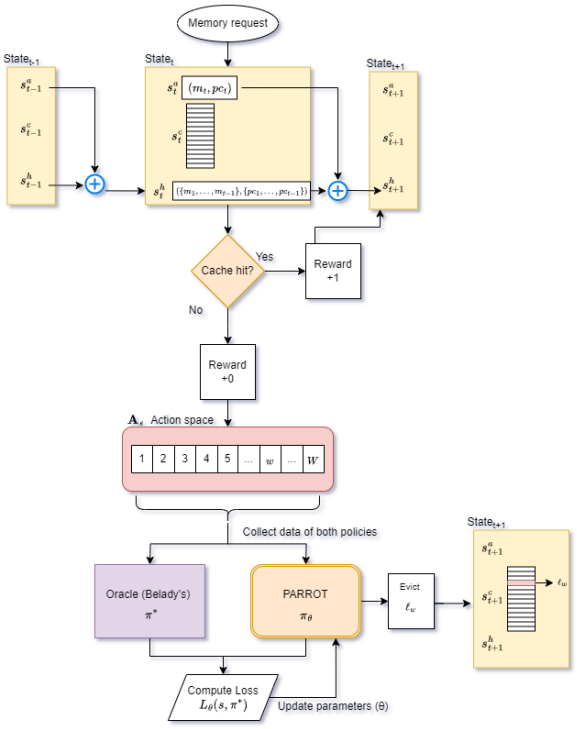

Fig. 4: The overall system diagram of the cache replacement problem cased as imitation learning when applying the Markov decision process

B. RAG

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.

C. Benchmarks Summary

1) 400.perlbench: The perlbench benchmark uses the programming language of ANSI C and is a simplified version of the popular scripting language, Perl v5.8.7. Additionally, it uses several third party modules. The inputs of this benchmark consists of three scripts. SpamAssasin is an Open Source spam checking software and acts as the primary script of this workload. This software takes care of scoring known copora of both spam and non-spam and provides samples of mail that is generated from inputted random components. Another input component is MHonArc which is a popular freeware that converts emails to HTML. The last script is a modified version of the "specdiff" script, which originates from the CPU2006 tool suite. The output of this mail-based benchmark is a line of summarizing salient characteristics. These characteristics may include the quantity of header lines or body lines.

2) 401.bzip2: The bzip2 benchmark is compatible with the programming language of ANSI C and is based on Julian Seward's zip2 version 1.0.3., Compared to bzip2 1.0.3, the SPEC's version of this benchmark doesn't perform file I/O except for when reading input. Compression and decompression from this benchmark only happens in memory and not in the CPU. There are six inputs for this workload: two JPEG images, a program binary, source code in a tar file format, an HTML file, and a "combined" file representing an archive of highly and not very compressible files. Each set of inputs are compressed and decompressed at different compression levels. The result of this benchmark is an outline of the entire process the benchmark is executing. The output sizes of each compression and decompression are also given.

3) 403.gcc: This gcc benchmark is based on gcc version 3.2. where it generates code for an AMD OPteron processor. It basically runs on a compiler where several of its optimization flags are raised and enabled. The nine input workloads for this benchmark are preprocessed C code files in .i file format. These files are: (1) cp-decl.i and expri.i from source files of 176.gcc from CPU2000. (2) 166.i from Fortran source files of a SPECint2000 benchmark. (3) 200.i which originates from a version of SPECfp2000 benchmark 200.sixtrack (4) Scilab.i from a version of the Scilab program (5) Expr2.i and c-typeck.i from the origin of 403.i (6) g23.i from the fold-const.c file from 403.gcc (7) s04.i comes from the sched-deps.c file of the 403.gcc benchmark

The output from these workloads consists of files of x86-64 assembly code.

4) 429.mcf: This mcf benchmark derives from MCF, which is a program that assists in vehicle scheduling in public mass transportation. For the benchmark's input, the number of timetabled and dead-head trips is on the first line of the input file, then a time time of its starting and ending time and for each dead-head trip, its starting and ending time and the cost of the trip. Note that the memory requirement of this benchmark requires approximately 860 and 17 megabyte for a 32 and 64 bit model respectively. When this benchmark is applied, there are two output files given and named inp.out and mcf.out. Inp.out contains log information and mcf.out has the check output values that describe a proposed, more optimal schedule.

5) 445.gobmk: The gombk benchmark plays the traditional game of Go and analyzing the positions of the Go pieces, using a set of commands from the program. The input if commonly given to the program in "SmartGo Format" (.sgf) which is a often used for standard representations of Go games. The output of this benchmark is a ASCII description consisting of a sequence of moves you can execute in the game.

6) 456.hmmer: In simpler words, the hmmer benchmark applies a technique to execute sensitive database searching as well as protein sequence analysis. The inputs for this benchmark are a database called sprot41.dat and nph3.hmm as a reference and input workload respectively. And the output consists of a ranked list of matches in the database search or sequence analysis for proteins.

7) 458.sjeng: The ssjeng benchmark is based on Sjeng 11.2, which is a program that can play chess and attempts to find the most optimal moves by primarily using priority proof number tree searches. The input of this benchmark includes a text file that has different chess positions in standard Forsyth-Edwards Notation (FEN) and how much this position should be prioritized and analyzed. The result that comes out of this benchmark is the optimal position the chess piece needs to be at as well as additional side information for context and the tree searching module that was used.

8) 462.libquantum: Libquantum is known as a library to simulate that of a quantum computer. Therefore, this benchmark gives some sort of structure that contains a quantum register and elementary gates. This benchmark program anticipates a number to be factorized as a command-line parameter. Additional parameters can be extra inputs to specify the base for the modular exponentiation portion in Shor's algorithm. As a result, the output provides an explanation of what occurred during executions and gives the conclusion of whether or not the factorization was successful.

9) 464.h264ref: This benchmark program is based on version 9.3 of the h264avc(Advance Video Coding) video compression software. The reference workload for input required includes two different, raw/uncompressed video data in YUV-format. To go more into the specifics, these files are foreman_qcif.yuv which is a standard sequence in video compression and sss.yub which is a sequence exported from a video game. As a result of these inputs, the executed run provides size, verified files which include encoded logs from the baseline and main profiles, the SS sequence and a leakybucketparam.cfg file for each input.

10) 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..

11) 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.

12) 483.xalancbmk: The xalancbmk benchmark is a modified version of Xalan-C++ which is a XSLT processor that uses a subset of C++.This processor is used for transforming XML documents into other file formats such as HTML files, text, or other XML document types

if requested. The input reference workload is simply an XML document and an XSL Stylesheet. With these input files, an HTML document, text file, or other XML-type document is outputted.

13) 410.bwaves: This bwaves benchmark simulates blast waves in a three dimensional transonic transient laminar viscous flow. The input workload includes grid size, flow parameters, the initial boundary condition and the number of time sets. Three data sets are also included: test, train and ref. These data sets differ based on their grid size and number of time steps. Because of the nature of the simulation, it is a challenge to determine validation. As a result, there are three different outputs which include a L2 norm of dq(l.i.j.k) vector after the last time step, the residual for convergence after every time step and the total sum of iterations for convergence for each time step.

14) 416.gamess: This gamess benchmark can do a wide range of various quantum chemical computations. Information regarding a description of its inputs and outputs could be found in the input.txt, intro.txt and prog.txt files from SPEC CPU2006.

15) 433.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.

16) 434.zeusmp: This benchmark is based off of a software called ZEUS-MP that involves computing fluid dynamics. The input files for this benchmark include a zmp_inp file that holds information regarding the parameters of the inputted problem such at mpitop (the arrangement of processors), pcon (problem control) and pgen (problem generator control information). Once this benchmark completes computation, the output we receive is a file called "tsl000aa" which has information for validation and about the blast wave from both the start and end of the fun.

17) 435.gromacs: This benchmark is derived from a package that constraints computations for molecular dynamics called GROMACS. The input can contain files with the setup for the inputted problem but test takes 1500 steps, train takes 3000 steps and ref takes 6000 steps. gromacs.out is the output text file that contains energy terms for validation. Note that due to the complicated and chaotic process that molecular dynamics may involve, the final computed values may slightly vary but should not vary by more than 1.25 percent from the reference values.

18) 436.cactusADM: This benchmark is a combination of an open source problem solving environment called Cactus and a computational kernel of numerical relativity called BenchADM. Primarily, CactusADM can compute Einstein's evolution equations in ADM 3+1 formulation. The workload required is a parameter file called BenchADM.par where grid size, number of iterations the code runs and time information if necessary are defined. For validation, the output contains each iteration, time and gxx and gyz components are printed and given in the output file.

19) 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.

20) 444.namd: This benchmark is derived from a parallel program for simulating large bimolecular systems called the NAMD. We use namd.input and the test, train and ref files as our input. Note that the namd.input file contains an atom simulation and this specific file format is created by NAMD 2.5. Outputting from this simulation using a command called "–output namd.out", creates an output file called namd.out and it contains calculations from the simulation and can be used for validation.

21) 447.dealII: This benchmark uses a C++program library called deal.II that is primarily used for the development of modern finite element algorithms as well as error estimators and adaptive meshes. In application, it can manage grid handling and refinement, degrees of freedom, and input of meshes and output of results in the format of graphics. Unlike some other benchmarks, the input is actually generated by code within the benchmark itself, which is a typical approach when it comes to

working with finite element applications. The output provides a description of the problem state at each step of refinement. To optimize the quantity of output the benchmark simplifies the solution to a caesar grid.

22) 450.soplex: This soplex benchmark is derived from a linear program that can run the Simplex algorithm called SoPlex, specifically Version 1.2.1.. The input files for this benchmark can be either in MPS or CPLEX LAP file format. These data files can be found from public domain sources and contain transportation planning models. The .out files that come out of this benchmark contain information for the optimal solution or when the iteration limit in the program has been reached. Any additional output files are under .stderr files and can contain the total number of iterations and how many of them were used.

23) 454.calculix: The calculix benchmark is derived from a software used for linear and nonlinear three-dimensional structural applications. A mesh that describes the geometry of the structure, material properties, geometric boundary conditions and natural boundary conditions is used as one of the inputs into the benchmark. Several keywords are also included and the reader can refer to a CalculiX document to get a better understanding of what each keyword may represent. As a result, variable fields across the structure such as displacements and stresses are what make up the output. The results of displacement are placed in the .dat file.

24) 459.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 "Filenamebase". 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.

25) 465.tonto: This benchmark came from an open source quantum chemistry package called Tonto and its objects involve simplicity and portability for quantum chemistry coding. A crystal structure alongside its atom positions, basis functions and X-ray diffraction data are included in the input data. Additionally the input provides calculation parameters wherever necessary. A main output file is given but updated on a regular basis to inform the reader how close the calculation is from the final answer. Once the last model wavefunction is acquired, we compare the X-ray diffraction data to the data from the benchmark executing.

26) 470.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.

27) 481.wrf: This wrf benchmark is based off of the weather prediction system called Weather Research and Forecasting (WRF) Model. The data sets used for input of this benchmark need to be created by the WRF Standard Initialization (SI) software. And as a result, from these datasets, we print out the temperature at a specific location on the grid at every time step and ensure to validate its values.

28) 482.sphinx3: The sphinx3 benchmark is based on a speech recognition system developed from Carnegie Mellon University. The input put into this speech recognition benchmark comes from files in the AN4 Database from CMU. The raw audio files from this database are either in big or little endian form. We can determine correct voice recognition in the output based on the examination of where utterances were recognized and identified. A trace of the language and acoustic scores are also provided.

Benchmark	Programming Language	Application of Benchmark
400.perlbench	C	PERL Programming Language
401.bzip2	C	Compression
403.gcc	C	C Optimizing Compiler
429.mcf	C	Combinatorial Optimization
445.gobmk	C	AI GO Game Playing
456.hmmer	C	Searching gene sequence database
458.sjeng	C	AI Chess Game Playing
462.libquantum	C	Physics/Quantum Computing
464.h264ref	C	Video compression
471.omnetpp	C++	Discrete Event Simulation
473.star	C++	Path-finding Algorithms
483.xalancbmk	C++	XML Processing
410.bwaves	Fortran	Fluid Dynamics
416.gamess	Fortran	Quantum Chemistry
433.milc	C	Quantum Chromodynamics
434.zeusmp	Fortran	Physics / CFD
435.gromacs	C/Fortran	Biochemistry/Molecular Dynamics
436.cactusADM	C/Fortran	Physics/General Relativity
437.leslie3d	Fortran	Fluid Dynamics
333.namd	C++	Biology/Molecular Dynamics
447.dealII	C++	Finite Element Analysis
450.soplex	C++	Linear Programming, Optimization
453.povray	C++	Image Ray-tracing
454. calculix	C/Fortran	Structural Mechanics
459.GemsFDTD	Fortran	Computational Electromagnetics
465.tonto	Fortran	Quantum Chemistry
470.lbm	C	Fluid Dynamics
481.wrf	C/Fortran	Weather Prediction
482.sphinx3	C	Speech Recognition

TABLE I: Summary of Benchmarks: includes benchmark name, programming language used and the common application of the benchmark.

D. Benchmark Overview and Discussion

Table 1 displays an overview of all twenty-eight benchmarks that have been summarized in the previous subsection. Approximately fifty-three percent of these benchmarks work with programming in C. In contrast, about thirty-five percent of the benchmarks work in Fortran. Fourteen percent overlap and work in both C and Fortan. Then there is a twenty-five percent portion of these benchmarks that run on C++. Looking at these proportions, C is in practice for the majority, while C++ is significantly a minority in contrast. Therefore, there is room for improvement regarding the balance in all the provided benchmarks. Now considering the applications of these bookmarks, there is a noticeable amount of diversity ranging from optimizers, physics simulations, data prediction and analysis, computing, and algorithms. Considering this variety, there is a strong amount of

coverage throughout these benchmarks in terms of usage and usefulness, but there are definitely still applications for which more benchmarks can provide testing and more efficient computing.

E. Prior works related Cache Replacement using Machine Learning

For this project, one of our most important references was the GLIDER cache replacement policy introduced by Zhan Shi et al. [3]. Deep learning models, while highly accurate, are often too large and slow to be directly implemented in hardware for cache prediction. However, the *GLIDER* cache replacement policy attempts to address this limitation by leveraging a powerful LSTM model offline to analyze cache behavior and extract key insights. These insights are then distilled into a lightweight, real-time online model that achieves performance similar to the LSTM but at a fraction of the computational cost.

GLIDER is evaluated on 33 memory-intensive workloads from SPEC 2006, SPEC 2017, and GAP benchmarks. It reduces cache miss rates by **8.9%** compared to LRU in single-core settings, outperforming state-of-the-art policies like *Hawkeye* (7.1%), *MPPPB* (6.5%), and *SHiP++* (7.5%). On a four-core system, it improves IPC (Instructions Per Cycle) by **14.7%** over LRU, exceeding gains from *Hawkeye* (13.6%), *MPPPB* (13.2%), and *SHiP++* (11.4%).

The design of the GLIDER cache replacement policy begins with an unconstrained offline caching model using an LSTM with an attention mechanism. This model identifies important program counters (PCs) in the input sequence, significantly outperforming the state-of-the-art Hawkeye predictor. The LSTM model takes a sequence of load instructions as input and predicts whether each load should be cached. It consists of three layers: an embedding layer, a single LSTM layer, and a scaled attention layer. The embedding layer converts one-hot encoded program counters (PCs) into learnable representations, the LSTM learns caching behavior, and the attention layer identifies correlations among PCs at each time step.

Through offline analysis of the attention layer, it is discovered that caching decisions rely primarily on a few critical memory accesses rather than the entire ordered sequence. The LSTM-based model for cache replacement relies on a sequence of past program counters (PCs) to predict whether each load instruction should be cached. Analysis of the attention mechanism reveals that while a long history of PCs (e.g., the past 30 PCs) is valuable, the caching decision does not depend on the precise order of this sequence. Instead, only a few critical memory accesses significantly influence predictions. The model uses a scaled attention mechanism, which improves sparsity in attention weights, helping identify these important PCs while maintaining high accuracy. Without scaling, the attention weights are uniformly distributed and less informative.

The findings highlight that accurate predictions can be achieved by focusing on a few key PCs, making the order of the sequence largely irrelevant. This is confirmed by shuffling the input sequence, which results in only marginal accuracy loss. These insights simplify the problem from sequence labeling to binary classification, where the presence of key PCs is sufficient for making effective caching decisions.

Further investigation maps these influential PCs to their source code, showing that the model learns application-specific semantics. This enables the identification of distinct caching behaviors for different memory accesses, providing a deeper understanding of the high-level semantics that drive optimal caching strategies.

Building on these insights, a practical online model is developed using an SVM trained to detect the critical PCs. This SVM achieves near-LSTM accuracy while being lightweight and efficient, suitable for real-time hardware deployment. The online model functions effectively as a perceptron for simplicity and practicality. This work highlighted the importance of studying attention layers and gave us an idea of what should expect from the attention layers of PARROT introduced by Liu et al. [2].

PARROT approaches cache replacement as an imitation learning problem framed within a Markov Decision Process (MDP). The objective is to learn a policy that maximizes cache hits by imitating optimal caching behavior.

The system's state at any time is defined by three main components:

- **Current Access:** Information about the memory address being accessed and its associated program counter.
- **Cache State:** The contents of the relevant cache set, such as which lines are currently stored.
- **Access History:** A record of past memory addresses and program counters, typically limited to a fixed number of recent accesses for training.

On a cache miss, the policy selects a line to evict, replacing it with the new memory line. On a cache hit, no eviction is required. Rewards are simple: a cache hit receives a reward of 1, while a miss receives 0.

State transitions are influenced by the program's memory access patterns and the policy's decisions. The goal is to approximate Belady's algorithm, which minimizes cache misses by using future access information (unavailable in real-time). By imitating Belady's behavior, the policy maximizes cumulative rewards over a sequence of memory accesses.

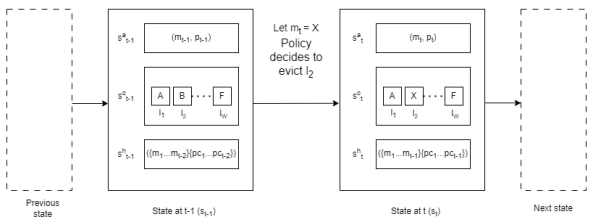

Fig. 5: PARROT Markov Decision Process

This framework allows the system to learn effective caching decisions based on historical patterns, even without explicit knowledge of future memory accesses.

The model architecture utilizes an LSTM to process the current cache access, $s_t^a = (m_t, pc_t)$, by generating embeddings for both the memory address ($e(m_t)$) and program counter ($e(pc_t)$). These embeddings are passed through the LSTM, producing hidden states for the past H timesteps, $[h_{t-H+1}, \ldots, h_t]$, which serve as keys for an attention mechanism. For each cache line l_w in the current cache state, a context vector g_w is computed by attending to the hidden states using the embedding of l_w as the query. The attention mechanism effectively prioritizes relevant information from the history.

The context vectors g_w are then passed through a dense layer followed by a softmax function to generate the policy, $\pi_\theta(a_t = w \mid s_t)$, which is a probability distribution over actions. The model selects the action with the highest probability, $\arg\max(\pi_\theta(a \mid s_t))$, determining which cache line to evict. This architecture leverages the combination of LSTM for sequential learning, attention mechanisms for identifying critical memory interactions, and dense layers for decision-making, allowing it to model complex caching behaviors efficiently.

The training process begins by transforming a sequence of cache accesses, $(m_1, pc_1), \ldots, (m_T, pc_T)$, into corresponding states $s_0, \ldots, s_T$. Optimal actions for these states are then computed using Belady's algorithm, forming a dataset $B = \{s_t\}_{t=0}^T$. At the start of training, Belady's algorithm is used to make cache replacement decisions, as the PARROT policy is not yet trained.

During training, batches of contiguous states, $s_{l-H}, \ldots, s_{l+H}$, are sampled. The LSTM hidden state is initialized using cache accesses from s_{l-H} to s_{l-1}. The PARROT policy π_θ is applied to the remaining states, $s_l, \ldots, s_{l+H-1}$, and a loss function, $L_\theta(s_t, \pi^*)$, is computed to encourage the learned policy to imitate Belady's decisions.

Every 5000 steps, a new dataset of the most recently visited states, $B = \{s_t\}_{t=0}^T$, is collected using the current PARROT policy, ensuring the training data reflects updated policy behavior.

The overall loss function described in the paper consists of two types of loss - Ranking Loss and Reuse Distance Loss. Thus, combined loss is given as:

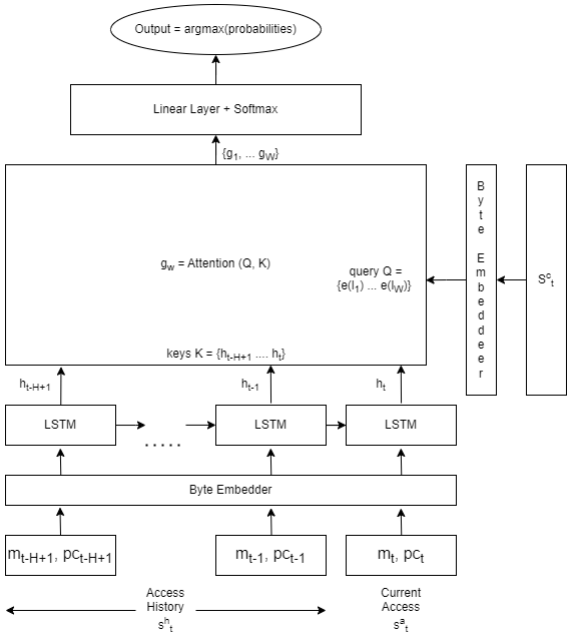

Fig. 6: PARROT Neural Network Architecture

$$L_\theta(s, \pi^*) = L_{\text{rank},\theta}(s, \pi^*) + L_{\text{reuse},\theta}(s, \pi^*).$$

The ranking loss for PARROT is an approximation of normalized discounted cumulaive gain (NDCG).

$$\mathcal{L}_\theta^{\text{rank}}(s_t, \pi^*) = -\frac{\text{DCG}}{\text{IDCG}}$$

To calculate the ranking loss we first calculate the differential approximation of the ranks for the lines in the cache state

$$\text{pos}(l_w) = \sum_{i \neq w} \sigma\left(-10(\pi_\theta(i \mid s_t) - \pi_\theta(w \mid s_t))\right)$$

Then we calculate the DCG using the predicted positions and true reuse distances

$$\text{DCG} = \sum_{w=1}^{W} \frac{d_t(l_w) - 1}{\log(\text{pos}(l_w) + 1)}$$

The IDCG is calculated using the true positions, making it a normalizing factor which would limit the loss function between -1 and 0

$$\text{IDCG} = \sum_{w=1}^{W} \frac{d_t(l_w) - 1}{\log(\text{ideal\_pos}(l_w) + 1)}$$

To calculate the Reuse Distance Loss, an additional fully-connected output layer in the PARROT network that processes the context embeddings g_w for each cache

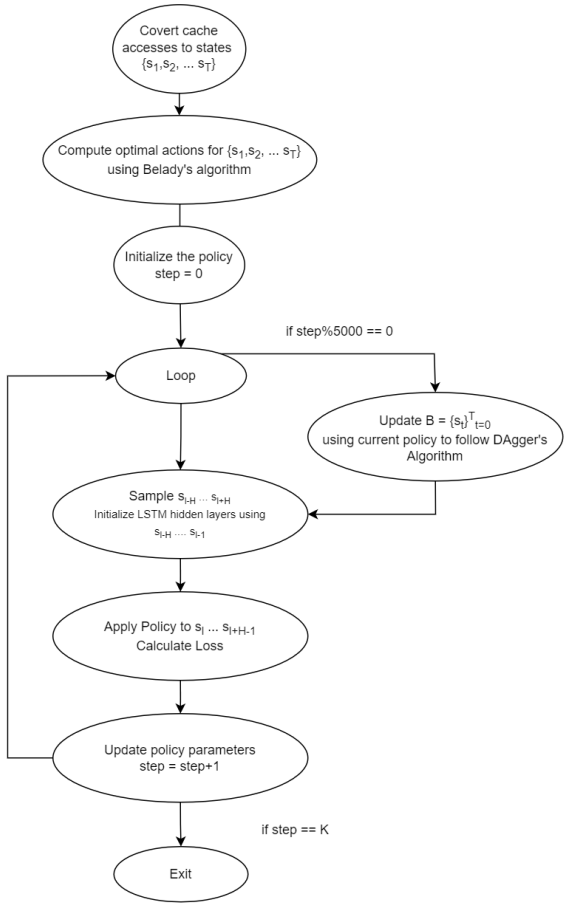

Fig. 7: PARROT Training Algorithm

line and generates predictions for the logarithm of the reuse distance $\hat{d}(g_w)$. This auxiliary head is trained using a mean-squared error loss function defined as:

$$L_{\text{reuse},\theta}(s,\pi^*) = \frac{1}{W}\sum_{w=1}^{W}\left(\hat{d}(g_w) - \log d_t(l_w)\right)^2.$$

We use the PARROT cache replacement project as a base for conducting all our experiments.

F. Feature Engineering

Yoo Et. al. [7] present a reinforcement learning based approach to cache replacement with a different interpretation of Belady's optimal algorithm (Belady+). The proposed method calculates the reuse distance for the accessed cache line along with the eviction candidates. This enables the policy to dynamically adapt to workload patterns in practical settings. The authors perform deep analysis to carefully the input features as shown in Fig. 8.

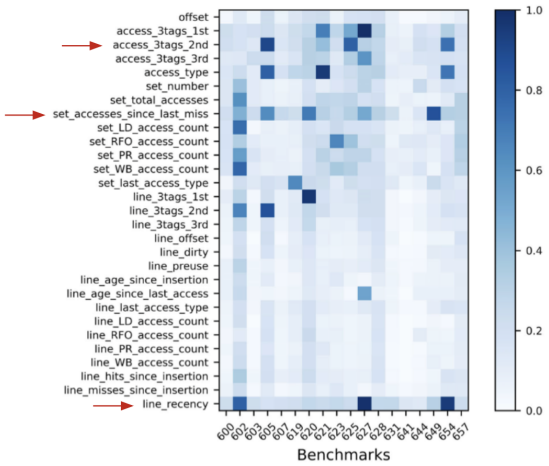

Fig. 8: Heatmap of weights for Victim Eviction by Yoo Et al.

Sethumurugan Et.al. [5] explore an RL driven cache replacement policy designed to strike a balance between computational cost and performance. Using lightweight ML models, the proposed approach uses key features such as recency, frequency, and spatial locality to predict optimal cache replacement decisions as shown in Fig. 9. The policy performs well across diverse workloads, achieving near-optimal hit rates and reduced latency.

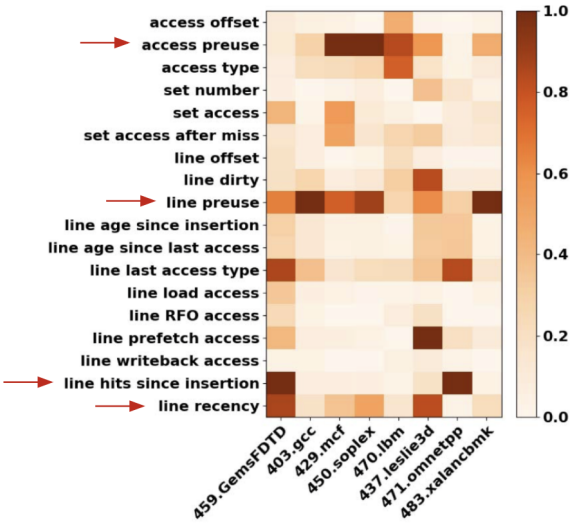

Fig. 9: Heatmap of weights for Victim Eviction by Sethumurugan Et al.

III. Motivation

A. Feature Engineering

The review of the literature carried out during Homework-2, evidently highlighted the importance of feature engineering and the choice of input features.

"Authors", all demonstrate in their results that the choice of input features strongly impacts the model's performance. Feature engineering could reduce storage overhead, model complexity, and improve workload interpretability. Improving the understanding of cache evictions and getting insights for cache replacement is the main motivation that we aim to explore using feature engineering.

B. High Level Semantics

Shi et. al. [6] performed a detailed analysis of the high level semantic understanding that Glider developed of the workload it trained and evaluated on. The basis for this was the omnetpp benchmark, and it was highlighted how memory accesses in the shadow of certain function calls are cache-friendly, while other function calls make the former cache-averse. Such insights are vital towards driving cache replacement decisions, especially if the policy is ML driven. An understanding of underlying program behavior baked into the neural-network would take the model a step further in generalizing its predictions across various workloads. This analysis could also be used to target and mitigate specific program behaviors that adversely affect caching behavior.

We extend this analysis such that it can be performed by an LLM agent that has specific knowledge about the benchmarks and replacement policies. **??** depicts an overview of the framework to enable this extended study. The objective is to feed the LLM with the high-level program source code of individual benchmarks, the associated assembly code with PCs and the cache eviction decisions taken at relevant PCs. To associate the assembly with the high-level code we use Ghidra an open-source disassembly and decompilation tool. For cache access traces, following the methodology used by [3], we use a combination of traces generated from the binaries which have the PC association, and existing traces used alongside ChampSim. The goal is to feed C3PU with all this data, and it should highlight potential semantic insights.

As a proof of concept, we attempt correlate the eviction decisions with program behavior for the mcf benchmark. Manual analysis of the relevant data reveals that a significant majority of misses are originate due to an array of pointers access pattern, similar to conclusions drawn by [1]. 11 shows a snippet of the code that is associated with this pattern and the disassembled PC for it. However, several limitations due to tool malfunctions impeded extending this to all workloads. The trace extraction tool used, DynamoRio, fails to generate a complete length of trace file, abruptly stalling due to run-time exceptions. Due to a lack of quality training and testing traces, that also correlate PCs to the disassembled binaries, the replacement policies can't

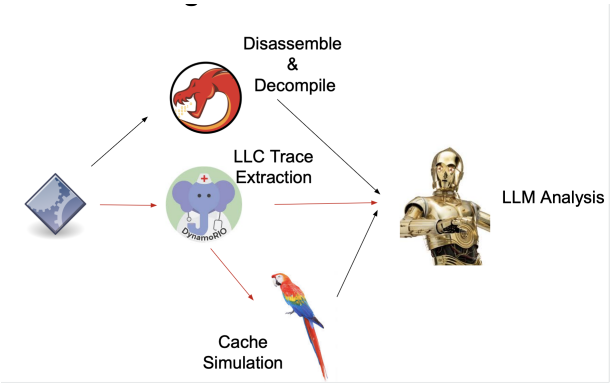

Fig. 10: Flowchart for Ghidra

give sufficiently accurate predictions that would lead to meaningful insights. An alternate approach is to use traces for training and testing the models that don't have correct PC correlations, like those collected from ChampSim. The decompiled assembly and program code can fed orthogonally, allowing the LLM to learn the underlying mapping between the trace PCs and decompiled PCs. Due to constraints on time, however, the latter approach is yet to be explored. This is a significantly open exploration that would give strong insights, motivating more interesting and better performing cache replacement policies.

```
arc_t *primal_bea_mpp( long m,  arc_t *arcs, arc_t *stop_arcs,
                       cost_t *red_cost_of_bea )
{
    long i, next, old_group_pos;
    ...
...
            /* red_cost = bea_compute_red_cost( arc ); */
0x4049d4    red_cost = arc->cost - arc->tail->potential + arc->head->potential;
            if( bea_is_dual_infeasible( arc, red_cost ) )
            {
...
```

[mcf] Highest count of accesses + misses : pointer chase accesses

0x4094d4 <-> 0x4037aa ?

Fig. 11: mcf code

C. LLM Fine-Tuning and Experiments

One of the main goals of our project was to explore the potential of open-source large language models, concretely the potential of the Llama series of language models from Meta, in improving neural-based cache replacement policies, more specifically the features fed into these neural network models.

Although the models proposed and described in [3] do exhibit good performance on many benchmarks, often beating the LRU replacement policy, the most commonly used cache eviction policy in the industry, we believe that these have significant room for improvement through the introduction of superior features, some that are more descriptive of the optimal cache line eviction decision.

Our approach centers around using and also fine-tuning the 3-billion-parameter Llama 3.2 model (`unsloth/Llama-3.2-3B-Instruct` on HuggingFace). There are two approaches as to how we gain insight into how better features can be designed.

The first approach deals with fine-tuning the above-mentioned model, making it an imitator of the Belady cache replacement policy, the theoretically best eviction policy there is. The rationale behind this idea is that if the LLM learns to imitate the Belady's eviction policy to a certain degree, it invevitably will have a better understanding of what could be improved (A parralel from the real world is that if somebody is good at something, he/she usually has more or better ideas what could be done to get even better). Thereafter, we prompt the fine-tuned model and ask it to identify better features that could lead to more informed and more effective cache eviction decisions. Again, the premise is that an LLM adept at performing cache eviction could also provide insights into features enhancing its decision making capabilities.

The second of our approaches to gain insight and understand cache-replacement better was to create an suite of tools for trace analysis, including a Retrieval Augmented Generation (RAG) system capable of retrieving data from multiple sources (e.g. a computer architecture textbook and a trace) and a system capable of analyzing structured data (which traces inherently are) by being adept at generating Python code. The results of our efforts are multiple systems, including `Vanilla RAG`, `Self-Consistent RAG`, `Multi-Source Self-Consistent RAG` and finally `Llama-The-Data-Scientist`, a system capable of analyzing enormous traces via natural language (for this system, small context windows are no longer the bottleneck). We have built our RAG system upon our RAG implementation from homework 2, adding system, user and assistant prompts, improving upon the usage of special tokens and finally reducing the unberable levels of abstractions, making tweaking and future improvements ever-easy.

IV. Design

A. Feature Engineering

Feature engineering consists of preprocessing raw data to extract useful and significant data, or features, that can allow for deeper insights or important projections of the raw data. Strategic feature engineering has been shown to improve cache eviction policies [5] and [7]. Specifically, as seen in Fig. 8 and Fig. 9, Yoo Et al. [7] and Sethumurugan Et al. [5] found that reinforcement learning models heavily weighted preuse distance, line hits since insertion, line recency, subset of 3 tag bits and access type features during training, suggesting that they are more significant for determining the cache line to evict.

Taking this into account, we extracted those features along with the cache and access parameters as a superset of features. We provided this superset of features to the RAG file and then trained Parrot on multiple subsets of features to try and improve its performance. The details of this are provided in the Methodology section V. The superset of features is explained in detail below:

- PC: program counter value for current access.
- Memory Address: full memory address of the current access
- Tag: Tag bits of the memory address used to identify a cache line
- block offset: gives the particular byte of data accessed in a cache line (not relevant for cache replacement).
- Set: cache set accessed by current request.
- Way: cache way accessed by current request.
- Access type: cache access type (LD, RFO, PF, WB)
- Line recency: shows how recently the cache line was requested by the processor.
- Hit/Miss: if the acccess was a hit/miss as per LRU policy for LLC.
- 3Tags1/2/3: 3 subsets of lower 9 bits of the tag ([25:23][22:20][19:17] of memory address)
- Hits since last insertion: number of times the cache line was requested by processor since it was inserted, i.e, brought into the cache.
- Set Accesses since Last Miss: Number of times the current set was accessed since last miss.

Although the results of the cited papers show promise in the choice of the selected features, it is important as CPU architects to reason about why they might be performing well. The aim of any of these features is to evaluate the utility of the cache lines present in the set and evict the ones with the lowest utlility when the need arises. Let us analyse the features in this regard:

- Line recency: shows how recently the cache line was requested by the processor in a particular set. Value lies in the range(0,#ways-1). This feature gives utility relative other lines in the set.
- Hits since last insertion: number of times the current cache line was requested by processor since it was inserted, i.e, brought into the cache. It different from recency in the sense that it keeps a count of the absolute number of hits not a relative count. This would give the direct utility of a line from the processor's perspective.
- Set Accesses since Last Miss: Number of times the current set was accessed since last miss. This is an indirect absolute measure of the utility of lines

within a set. Could be thought of as recency counter that doesn't saturate at #way-1.
- Type of access: Type of access could have a coerlation with the hit rate due to its nature. eg: a prefetched line could have a high probility of hitiing.
- 3Tags1/2/3: This is the least intuitive feature and we C3PU could provide insights into this feature.

B. 4-bit Quantized Fine-tuning with QLoRA

QLoRA (Quantized Low-Rank Adaptation) is an advanced fine-tuning method for large language models (LLMs) that drastically reduces memory requirements without sacrificing performance. This technique allows for fine-tuning large models on consumer-grade hardware, making it accessible to researchers and developers with limited computational resources.

1) Core Elements of QLoRA: **`1. 4-Bit Quantized Base Model`** QLoRA employs a pre-trained language model quantized to 4-bit precision, significantly lowering memory usage compared to full-precision models. This aggressive quantization is achieved through several key innovations:

1) **4-bit NormalFloat (NF4)**: A new data type designed to be information-theoretically optimal for weights that follow a normal distribution, offering better model quality retention than traditional integer quantization.
2) **Double Quantization**: This approach compresses the quantization constants further, further reducing the model's memory footprint.

`2. Low-Rank Adapters` To capture task-specific information during fine-tuning, small trainable modules are added to the frozen base model. These adapters utilize the LoRA (Low-Rank Adaptation) technique, which approximates full-rank matrices by using two smaller matrices.

`3. Paged Optimizers` A memory management strategy that efficiently swaps optimizer states between CPU and GPU memory, preventing memory spikes during training.

2) How QLoRA Functions:

1) The pre-trained language model is quantized to 4-bit precision and kept frozen.
2) Low-Rank Adapters are integrated into the model.
3) During fine-tuning, gradients are backpropagated through the frozen 4-bit model into the Low-Rank Adapters.
4) Only the LoRA layers are updated during training.

3) Advantages of QLoRA:

- **Memory Efficiency**: QLoRA significantly reduces memory usage. For example, fine-tuning a 65B parameter model is reduced from over 780GB to less than 48GB of GPU memory.
- **Competitive Performance**: Despite the aggressive quantization, QLoRA maintains near-equal performance to full-precision fine-tuning. It can reach 99.3% of ChatGPT's performance with just 24 hours of fine-tuning on a single GPU.
- **Increased Accessibility**: The reduced resource demands make LLM fine-tuning more accessible. For instance, QLoRA enables fine-tuning a 33B parameter model on a single 24GB GPU and a 65B parameter model on a single 48GB GPU.
- **Enhanced Efficiency**: QLoRA processes large data sequences more efficiently than traditional LLMs due to its query-based local attention mechanism.

TABLE II: Performance Comparisons

Methods	Bits	7B	13B	30B	70B
Full	16	60GB	120GB	300GB	600GB
LoRA	16	16GB	32GB	64GB	160GB
QLoRA	4	6GB	12GB	24GB	48GB

4) Performance Comparisons: QLoRA outperforms all previously released models on the Vicuna benchmark. It matches the accuracy of traditional LLMs while requiring far fewer computational resources. In some instances, fine-tuning QLoRA on a small, high-quality dataset can achieve state-of-the-art results, even with smaller models compared to the current field leaders.

V. Methodology

A. Feature Engineering

The authors of Parrot [3], provided the setup to extract pc and memory address for CRC2 trace files using the ChampSim simulator. We make changes in these files as shown in to extract pc, memory address, set, way, access type, recency and lru hit/miss. We then made changes to the train_test_split.py file to handle these additional features (columns in CSV files) and split them into train, valid and test CSV files as done with Parrot.

```
// called on every cache hit and cache fill
void CACHE::llc_update_replacement_state(uint32_t cpu, uint32_t set, uint32_t way, uint64_t full_addr, uint64_t ip, uint64_t victim_addr, uint32_t type, uint8_t hit)
void CACHE::llc_update_replacement_state(uint32_t cpu, uint32_t set, uint32_t way, uint64_t full_addr, uint64_t ip, uint64_t victim_addr, uint32_t type, uint8_t hit, u_int32_t
{
    // baseline LRU
    if (hit && (type == WRITEBACK)) // writeback hit does not update LRU state
        return;

    // Log PC, address pairs to create the trace
    //llc_access_trace << hex << ip << "," << full_addr << endl;
    llc_access_trace << hex << ip << "," << full_addr << "," << set << "," << way << "," << type << "," << dec << recency << "," << static_cast<int>(hit) << endl;

    return lru_update(set, way);
}
```

Fig. 12: Change in function to get features from ChampSim

We then wrote a python script that reads these traces and calculates the extra parameters, i.e, tag & block offset and the engineered features, i.e, hits since last insertion, set accesses since last miss and 3 tag bits for every access in the trace. These results are then appended to the traces extracted from ChampSim to

create our superset of features. Fig. 13 shows the code implementations of all these functions. The code files can be found on GitHub.

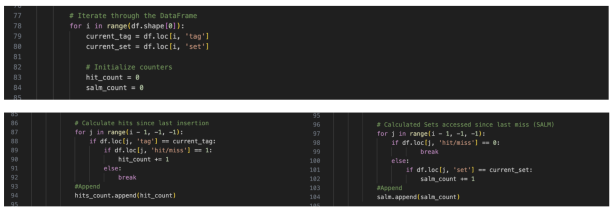

Fig. 13: Code snippets from Features.py

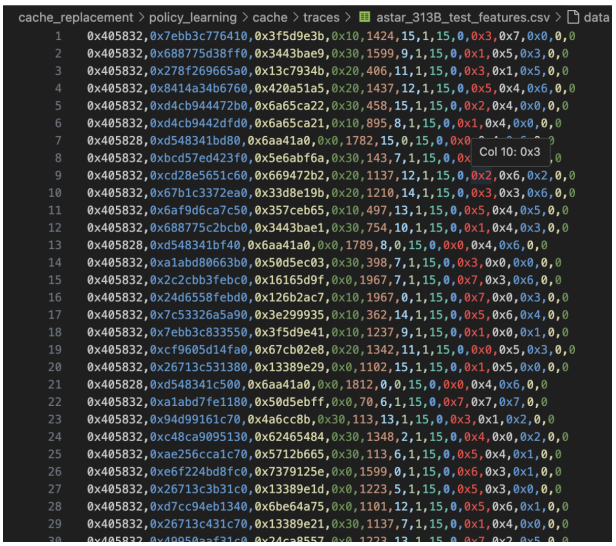

Fig. 14: Example trace file (Superset of features)

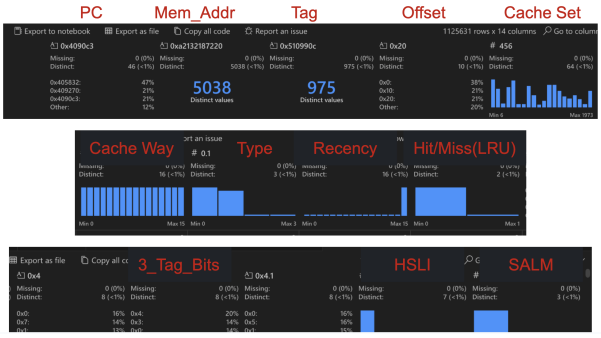

Fig. 15: Statistical plots used as aid in benchmark questions

The python script took less than an hour to extract features from astar benchmark but ran longer than 12 hours for some other benchmarks, limiting our experiments to only the astar benchmark. This could be due to a versioning error or due to the presence of nested loops and execution time of python programs. Using a C based cache simulator or the same script translated to C are two possible ideas we thought of to mitigate this execution overhead.

To get these additional features into PARROT, the codebase had to be changed to acquire the features from the trace CSV file and feed them into `model.py`. This process began in `memtrace.py` where the CSV file reader was altered to parse the CSV into each cache access' PC, memory address, and other features. For mobility convenience, these other features were kept in a list for each cache access and separated into individual features in `model.py`. After acquisition of the feature list from the CSV, it is then stored in a look-ahead-buffer, which the main script, `cache_model/main.py`, grabs from to create cache access objects. These cache access objects, with their newly acquired features, are then fed into `model.py` in batches, where they are processed and an eviction decision is made.

The following combinations of features were trained and evaluated on the astar dataset using the original PARROT architecture with the same testing parameters as stated in [4]. Each training was evaluated at its 20000 step checkpoint.

- PC, memory address, cache access set, cache access way, instruction type, and cache line recency.
- PC, memory address, hit count, and SALM.
- PC and 3tag1/2/3.

Furthermore, each set of features was trained and evaluated for four different embedding types:

- Integer: PC and memory address embedded with dynamic vocab embedder (64). Other features not embedded, rather fed in as integers.
- Dynamic: All features embedded with dynamic vocab embedder (64).
- Byte: All features embedded with byte embedder (64).
- SmallByte: PC and memory address embedded with byte embedder (64). Other features embedded with byte embedder (16).

```
memtrace.py:
    row = next(self._csv_reader)
    pc , address, features = row[0],
        row[1], row[2:]
cache_model/main.py:
    cache.read(pc, address, features, [
        add_to_data])
model.py:
    features = [cache_access.features
        for cache_access in
        cache_accesses]
```

In addition to the feature and embedding exploration and as an extension of Homework 2, the single layer

perceptron, 2 layer MLP, PARROT-RNN, and PARROT-LSTM models were trained on traces where the PC and memory address were XORed together instead of given separately. However, as mentioned in Homework 2, there was suspicion of a bug in the implementation of the single layer, double layer, and RNN PARROT models. An attempt was made to exterminate this bug by refactoring the original code and implementing these new architectures in their own environments. To achieve this re-factorization, completely new versions of the `cache` and `cache_model` directories were created for each new architecture to ensure there were no calls to the original model.

B. Attention Layer Analysis

To thoroughly analyze the attention layer of the PARROT model, we developed a tailored approach based on the attention visualization method used in GLIDER, with key modifications to accommodate the multi-dimensional cache line embedding of PARROT. Unlike simpler models, PARROT generates a 30x16 attention matrix, where 30 represents the access history and 16 corresponds to the cache associativity. This matrix is essential for understanding how the model learns eviction decisions based on the history of cache accesses and the program counter (PC).

The first step in our approach involved modifying the PARROT codebase to ensure it could output relevant metadata during training. Specifically, we made changes that enabled the model to dump crucial information, such as the PC, cache line address, cache state, access history, eviction scores, and the attention matrix, into a Pickle file. This metadata is pivotal for tracking the evolution of cache access patterns and their correlation with eviction decisions.

Once the relevant data was collected, we focused on visualizing the attention layer's behavior by selecting a target PC from the dataset and identifying the cache line that was evicted during that particular access. To focus the analysis, we isolated the specific way corresponding to the evicted cache line within the attention matrix associated with the target PC. This process was repeated for a total of 30 distinct target PCs, with each iteration contributing to an updated matrix. The resulting 30x30 matrix, representing the attention weights for the evicted cache line at various target PCs, was then visualized as a heatmap.

This visualization method is grounded in the hypothesis that the eviction behavior may be driven by certain patterns within the attention distribution. By focusing specifically on the attention weights associated with evicted cache lines, we can gain a deeper understanding of the decision-making process behind eviction, revealing potential regularities or correlations that the model has learned over time.

Furthermore, the Pickle file containing this detailed metadata can be converted into a JSON format. This conversion allows for easy integration with systems such as the RAG (Recurrent Attention Graph) system, enabling systematic feeding of the attention layer data for further analysis or post-processing. By structuring the data in this manner, we open up the possibility for additional insights, making it easier to apply advanced analysis techniques or enhance the overall learning process.

This method of analyzing the attention matrix provides a powerful tool for understanding the inner workings of the PARROT model, offering not only a visualization of its learned behavior but also a path toward further optimizations and refinements. We analyzed the attention layer for 19 different benchmarks. For a few benchmarks, we also varied the cache size to observe how that would affect the attention layers. The insights gained from these attention plots could lead to more efficient cache management strategies, improving the overall performance of the model in real-world scenarios.

VI. LLM Fine-tuning and Experiments

A. Vanilla RAG

In the Vanilla RAG system, we implement a question-answering system that will retrieve some relevant document chunks and generate an answer based on the context that is retrieved.

1) Process:

1) **Document Loading and Preprocessing**: Load and split the document into smaller chunks using RecursiveCharacterTextSplitter from the langchain library.
2) **Embedding Generation**: Use HuggingFaceEmbeddings to convert each document chunk into vector embeddings.
3) **Vector Store**: Use FAISS from langchain_community to create a vector store that enables efficient similarity search which can retrieve the document chunks related to the query.
4) **Question Answering**:
    a) When a question is asked, retrieve the top 'k' document chunks found in the vector store.
    b) Construct a prompt that contains the user question and the retrieved document chunks.
    c) Provide this prompt to the LLM to construct an answer.
5) **Answer Generation**: The language model uses the user query and the retrieved document chunks to formulate an answer relevant to the question.

B. Self-Consistent RAG

A Self-Consistent RAG system or a Self-Consistency RAG system is an advanced form of Retrieval Augmented Generation in which multiple answers are generated using the same document chunks and user query. These answers are compared and their consistency is evaluated. The most consistent parts of the answers are used to provide a more definitive answer.

1) Process:

1) **Configuration of the RAG System**: Declare the parameters of the Self-Consistent RAG system such as the number of documents to be retrieved, the Large Language model being used, the number of answers to be generated, and the randomness of the generated answer.
2) **Initialization of the Question-Answering System**: Set up the question-answering system using the given model, tokenizer, prompt, and documents. This process is similar to the one implemented in the RAG system.
3) **Self-Consistency**: Start the self-consistency loop and collect the answers for a specified number of steps.
    a) For each iteration, a system prompt is provided with the document chunks retrieved from the vector store and the user query. Each iteration will contain a different answer based on the provided temperature (randomness) assigned to the model.
    b) The collected answers are provided to a new system prompt that generates an answer, based on the majority of the collected answers.

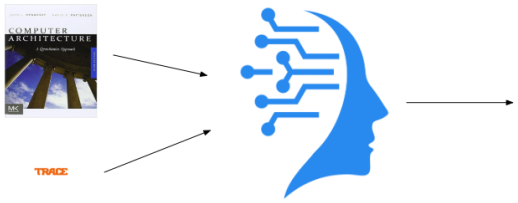

Fig. 16: Overview diagram of our RAG system

In Fig. 16, we can see the basic format of the RAG. We can apply this to variations of the RAG system such as the Vanilla and Self-Consistent RAG system as described above.

C. Multi-Source Self-Consistent RAG

The Multi-Source Self-Consistent RAG system is a variation of the Self-Consistent RAG system, in which multiple documents are used, rather than using a single document to answer the user query. This method ensures improved accuracy and contextual richness.

1) Process: The process of the Multi-Source Self-Consistent RAG system is very similar to the process of the Self-Consistent RAG system. The only difference between them is that there are multiple documents that are fed to the Vector Store, so that there is more context and varied outputs in the self-consistency loop.

D. Llama-The-Data-Scientist

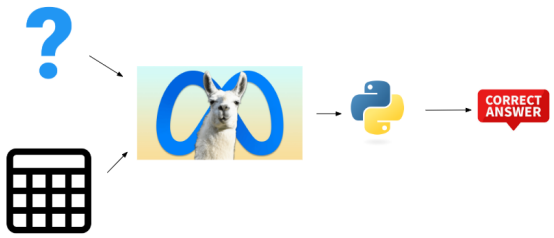

Fig. 17: Overview diagram of our `Llama-The-Data-Scientist` system

In Fig. 17, we see the basic modeling of our LLM. It starts by receiving a query and analyzing traces as inputs. Then we use our LLM as a python code generator for analyzing tabular data using pandas, leading us to our outputted answer.

VII. Results/Analysis

A. Feature Engineering

The misses per kilo instruction (MPKI) and normalized cache hit rate (NCHR) were evaluated and used as the performance metric for the eviction policies trained on the different feature/embedding combinations. The normalized cache hit rate metric is defined in [4] as

$$r_{normalized} = \frac{r - r_{LRU}}{r_{opt} - r_{LRU}},$$

where r is the raw cache hit rate of the trained eviction policy, r_{LRU} is the raw cache hit rate of LRU, and r_{opt} is the cache hit rate of the optimal eviction policy, Belady's. The results for the astar workload are shown in Fig. 18 and Fig. 19 and are summarized in TABLE III.

Feature Set	Embedding	MPKI	% Increase/Decrease MPKI	Normalized Hit Rate	% Increase/Decrease Hit Rate
Big4	Integer	30.608	1.56%	0.817	-1.57%
	Dynamic	30.206	0.23%	0.844	**1.68%**
	Byte	38.027	9.14%	0.330	-39%
	Small Byte	34.914	0.20%	0.537	-0.92%
HC&SALM	Integer	30.477	1.13%	0.826	-0.48%
	Dynamic	30.644	1.68%	0.814	-1.93%
	Byte	35.500	1.88%	0.496	-8.49%
	Small Byte	36.263	4.07%	0.444	-18.10%
3Tags	Integer	30.288	0.50%	0.843	**1.57%**
	Dynamic	30.247	0.37%	0.845	**1.80%**
	Byte	34.689	**-0.44%**	0.548	**1.12%**
	Small Byte	34.867	0.07%	0.540	-0.37%
PC&Address	Dynamic	30.137	N/A	0.830	N/A
	Byte	34.844	N/A	0.542	N/A

TABLE III: MPKI and Normalized Hit Rate for Different Feature Sets and Embeddings with Percentage Changes

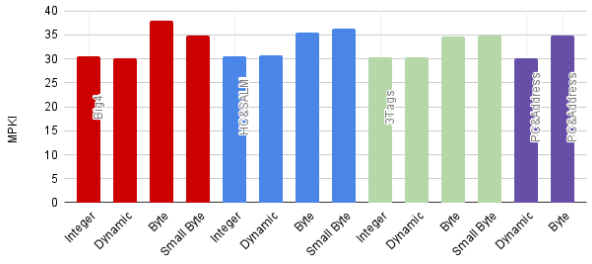

Fig. 18: MPKI for different feature/embedding combinations on the astar workload.

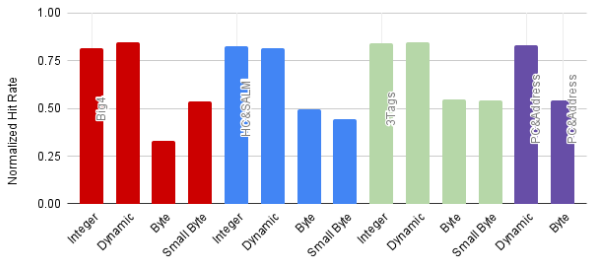

Fig. 19: MPKI for different feature/embedding combinations on the astar workload.

As can be seen, most of the feature/embedding pairs do not increase the normalized cache hit rate of PARROT except for a few select cases. These cases are Big4/Dynamic, 3Tags/Int, and 3Tags/Dynamic, which have an average increase of 1.5%. Otherwise, for the trainings where the PC and memory address are embedded with the dynamic embedder, the cache hit rate decreases by and average of 1.33% and 10.96% for byte embedding. The MPKI does not improve for any of the features except a small instance for the 3Tags feature set with byte embedding. The average MPKI increase is 1.7%.

Analysis of the feature engineering results shows that the Big4 and 3Tags features could have the ability of improving based cache eviction policy performance. However, regarding the implementability of cache eviction policies, we hypothesize and suggest that the 3Tags feature set, specifically with integer embedding, could prove to be a significant and non-computationally expensive feature for improving cache eviction policies. Not only does it show improvement in cache hit rate for the astar workload, but it reduces the input dimension size by a factor 95% as opposed to a length 64 embedding vector for the memory address.

We also note that often the integer embedding is competitive to dynamic embedding in MPKI and NCHR for most features. This could suggest that in future cases where additional features are utilized, lengthy embedding vectors and corresponding layer dimensions may be redundant, where a single neuron can achieve similar results with an integer. This idea is also extendable to comparing the byte embedder versus small byte embedder, where the length 16 small byte embedding vector is competitive with the length 65 large byte embedding.

A comparison of the MPKIs for the original data (PC and memory address) versus the XOR data trained on the four models used in Homework 2 can be seen in Fig. 20 and Fig. 21. Based on the graphs, the hashing of the PC and address with the XOR operation does not show a consistent change in performance for any of the models. Rather, all models achieve nearly equivalent performance on each of the workloads.

The results for the XOR data did not enlighten our original confusion from our Homework 2 results. In fact, the XOR data is even more monotonous. This trend is surprising, considering the PC feature is widely considered one of the most significant features for cache eviction, and seemingly abstracts this feature. It is even more surprising considering the code to implement the

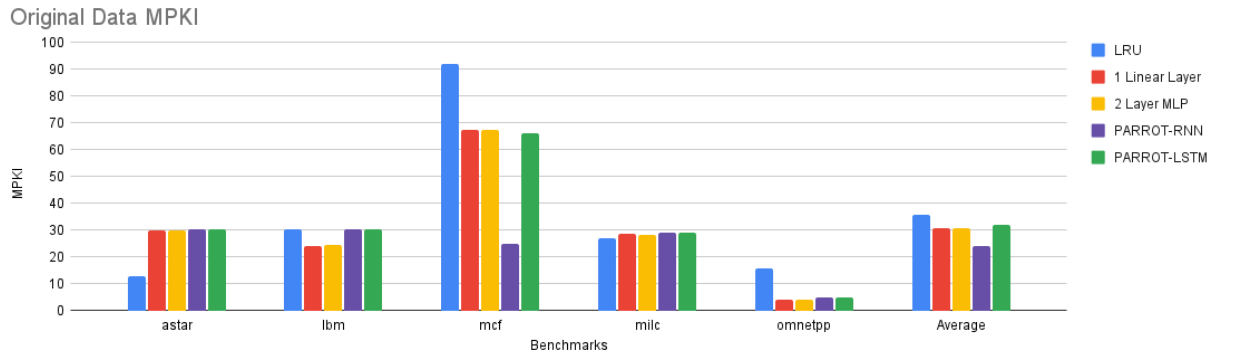

Fig. 20: MPKI for original features (PC and address) across multiple model architectures.

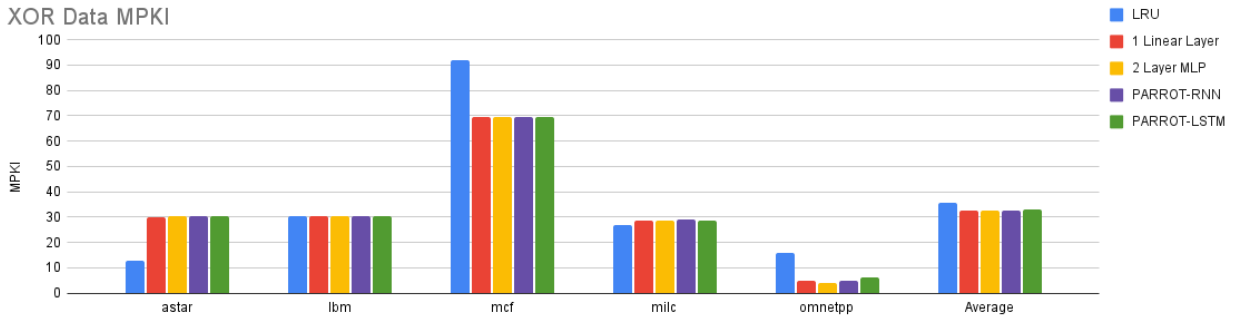

Fig. 21: MPKI for XOR PC and address across multiple model architectures.

different model architectures was refactored to be more robust. Our current hypothesis follows that which was suggested in Homework 2, which entertained the idea of PARROT being a bloated model that introduces too much complexity. If this is true, perhaps the hashing of two input features into one allows simpler models to achieve good results. Regardless, more research into this idea needs to be done for any conclusive results.

B. Attention Layer Analysis

In this subsection, we present the results of our analysis of the attention layer within the PARROT model, focusing on insights related to cache eviction decisions. Our findings corroborate key observations made by GLIDER [3], which highlighted that the presence of specific program counters (PCs) within the cache state plays a more significant role in determining eviction decisions than the sequence in which cache accesses occur. This aligns with the intuition that certain access patterns or program counters may be more influential in the eviction process, regardless of their position in the access history.

A central hypothesis of our analysis is that the way a cache line gets evicted may reveal discernible patterns within the attention matrix, providing valuable insights into the eviction behavior of the model. By isolating attention weights related to the evicted cache line, we sought to uncover whether the model's attention mechanism could help explain its decision-making process. Our results show a strong correlation across target offsets, indicating that the attention matrix contains systematic patterns that are consistent across different program counters. These results suggest that the attention mechanism is indeed learning to focus on specific aspects of the cache state, such as certain cache lines or PCs, which significantly influence eviction decisions.We have plotted selective benchmarks which show 3 patterns, low correlation, medium or mixed correlation or very high correlation. Regarding the benchmarks that we left out of this paper despite having the results, they show a medium or low correlation.

Low correlation traces do not show a very clear discernible pattern in the attention plots. The benchmarks namd, sjeng and sphinx3 fall under this category. The namd benchmark is a molecular dynamics simulation program widely used for biomolecular simulations, which involves complex, time-dependent interactions between a large number of particles. These types of simulations are highly dynamic, with cache accesses driven by irregular memory patterns and unpredictable

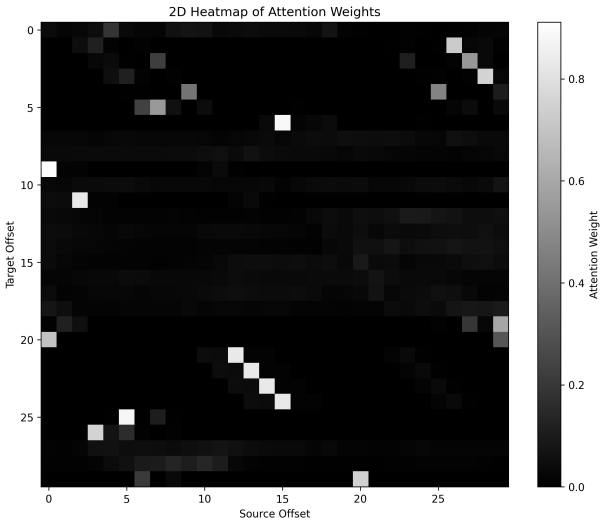

Fig. 22: namd

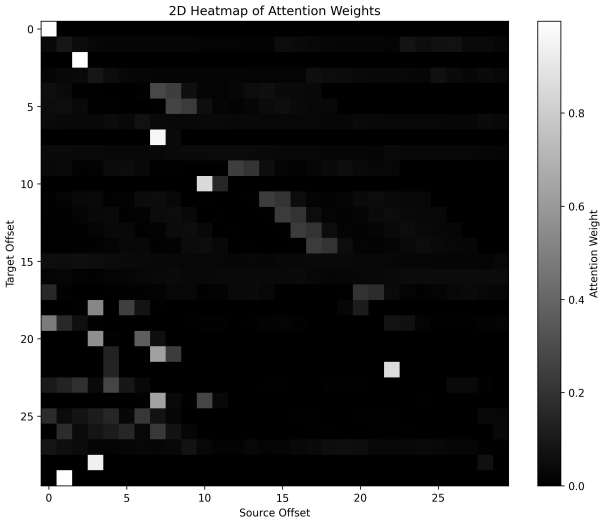

Fig. 23: sjeng

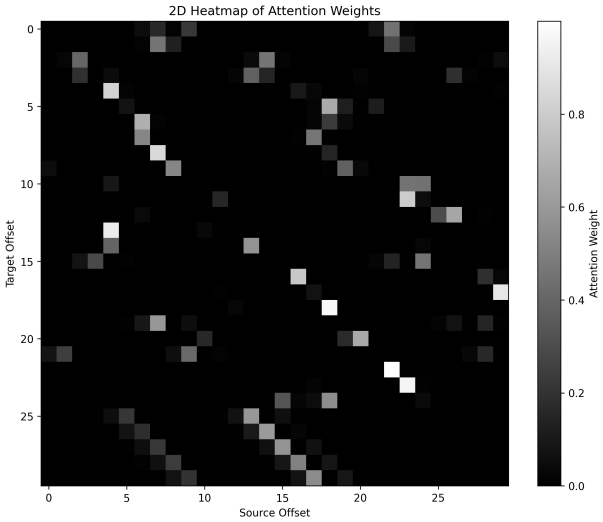

Fig. 24: sphinx3

Fig. 25: Low Correlation Traces

memory access behavior. As a result, the attention layer in PARROT might not show clear correlations because the cache behavior in namd is largely influenced by non-repetitive, sparse access patterns across a vast state space, which contrasts with workloads that exhibit more predictable memory access patterns.

The sjeng benchmark is a highly parallel chess engine that relies on a combination of deep search algorithms and evaluation functions to determine optimal moves. Given its memory access patterns, which are often influenced by both the game's search depth and the move generation process, sjeng exhibits a mix of regular and irregular cache access behaviors. The faint pattern observed in the attention layer of PARROT likely reflects correlations related to specific stages of the search process, where certain moves or positions have more predictable memory access patterns. However, the random noise in the bottom part of the attention layer can be attributed to the less structured memory access during deeper search levels or positions with more complex evaluations.

The sphinx3 benchmark is a speech recognition system that heavily relies on sequential processing and large-scale memory access due to its need to handle complex linguistic models and search for optimal word sequences. Given this, the memory access patterns in Sphinx3 are more spread out and less regular compared to more deterministic benchmarks. The faint pattern observed in the attention layer of PARROT reflects correlations that are likely tied to the dependencies between different PCs during speech decoding. However, the sparsity and distribution of the attention pattern across multiple PCs suggest that cache evictions are influenced by a broader range of memory accesses, often linked to the various stages of the speech recognition process. This distributed attention behavior may result from Sphinx3's parallelized structure and its need to access a wide range of memory regions during both feature extraction and decoding, making the eviction prediction more challenging.

Medium correlation traces show an almost clear pattern in the attention plots, bit it might be slightly noisy, or spread across a few PCs. The benchmarks gcc, perlbench and soplex fall under this category. The gcc benchmark is a widely used set of programs for evaluating the performance of compilers and computing systems. It involves tasks such as compiling code, optimizing functions, and handling large data structures. Given the computationally intensive and memory-demanding nature of this benchmark, the attention pattern observed in the PARROT model's attention layer reflects the sequential and somewhat predictable memory access patterns that emerge during compilation tasks. The clear pattern spread across several PCs suggests that certain

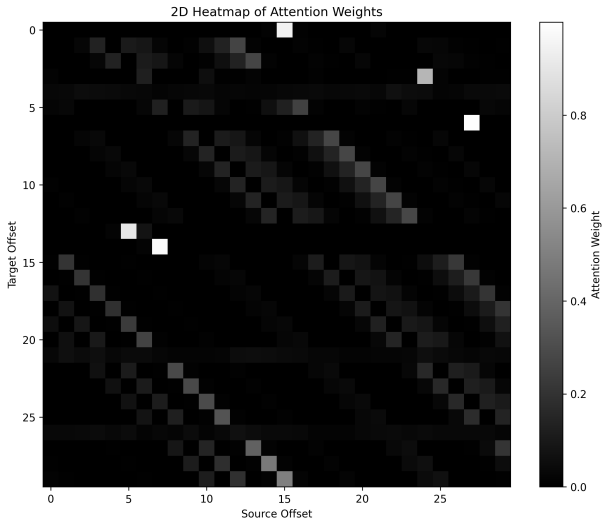

Fig. 26: gcc

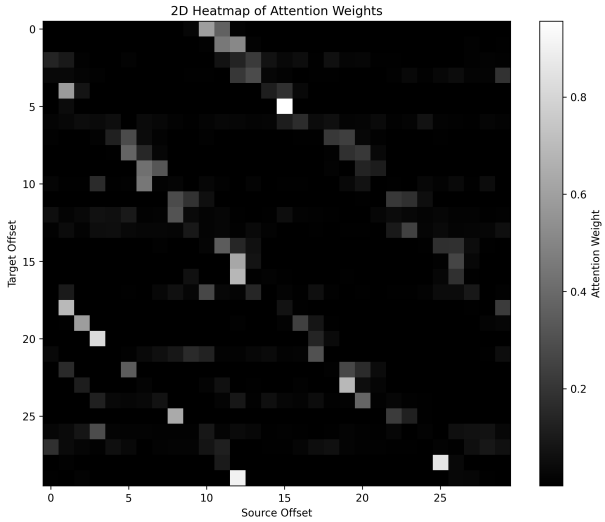

Fig. 27: perlbench

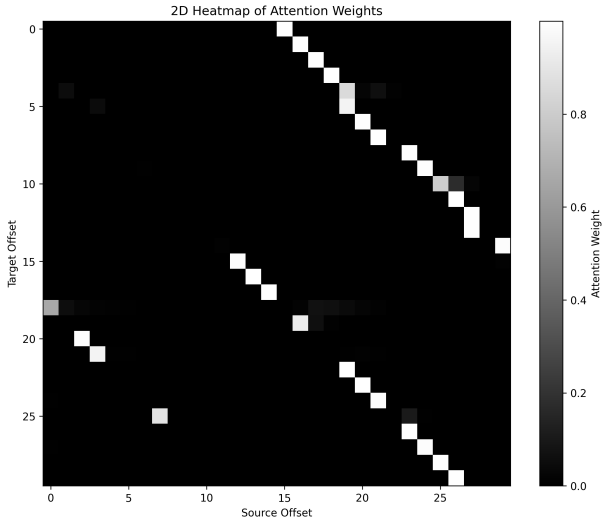

Fig. 28: soplex

Fig. 29: Medium Correlation Traces

memory accesses, such as those related to instruction parsing, function optimizations, or loops, have a more significant influence on cache evictions. The occasional random spots with high attention might correspond to less frequent but still important cache accesses, potentially tied to specific compiler optimizations or irregular memory access patterns within certain compiler phases.

The perlbench benchmark is designed to evaluate the performance of compilers and systems with a focus on Perl script processing, including tasks like regular expression handling and string manipulations. These operations typically involve complex and highly dynamic memory access patterns due to the nature of the Perl interpreter and its frequent data lookups. The attention pattern observed in the PARROT model for the perlbench benchmark shows a spread across 2-3 PCs, indicating that cache eviction decisions are influenced by a few critical memory accesses during the execution of the benchmark. The noisy and somewhat diffuse attention pattern suggests that while these program counters play a role in cache eviction, other less predictable or less significant accesses also contribute to the attention distribution. This could reflect the intricate and often irregular memory access behavior inherent in Perl's execution model, where certain memory accesses are more important at specific points in the script, but overall memory access is less consistent across the benchmark's execution.

The soplex benchmark is designed to evaluate the performance of solvers for large sparse linear systems, commonly used in scientific computing and optimization tasks. It typically involves a significant amount of matrix manipulation and iterative numerical methods, which lead to predictable, but sometimes intermittent memory access patterns. The attention pattern observed in the PARROT model for the soplex benchmark reveals a strong confidence in a single PC at the beginning of the execution. However, this attention focus shifts toward different PCs midway through the plot, suggesting that the model adapts to varying memory access patterns as the execution progresses, as the code moves to a new calculation. The model appears to be highly attentive to different memory regions at different stages, which could reflect the dynamic nature of the computations and data locality changes in the benchmark

The calculix benchmark gave is the most clear attention plot out of all our results. It simulates structural analysis problems, often involving large-scale finite element models. Its code frequently accesses memory in a highly structured manner due to the nature of iterative numerical methods, like solving sparse linear systems. These methods lead to predictable memory access patterns, particularly when matrices are repeatedly updated during computations. In the case of the attention layer plot for Calculix, the attention remains sharply

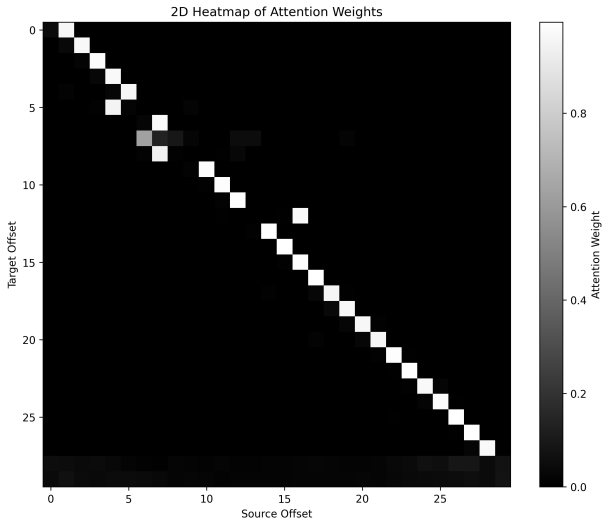

Fig. 30: calculix

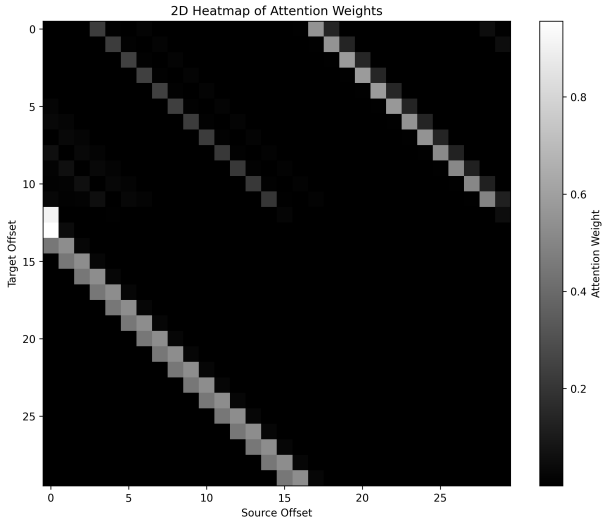

Fig. 31: zeusmp

Fig. 32: High Correlation Traces

focused on a single PC throughout most of the execution. This suggests that the PARROT model is identifying a key computational phase where memory accesses are concentrated, likely related to a specific loop or kernel in the solver. Given that the benchmark's memory access behavior is primarily influenced by this loop, the model learns to focus on this region, indicating that the eviction policy is being influenced by highly localized memory regions.

The zeusmp benchmark is designed to simulate a high-performance multi-user system running complex, real-time web services and database operations. This benchmark exhibits significant inter-process communication and a combination of random and structured memory access patterns, depending on the specific workload being simulated. In the attention layer plot for Zeus, a distinct pattern is observed, where attention is consistently focused on two PCs at a time, and the two it is attentive to change between them across different segments of the execution. This behavior likely reflects Zeus's underlying code structure, where certain phases of computation or specific threads of execution dominate at different points in the run. The attention switching between two PCs could correspond to different but equally important computational tasks, such as handling separate database transactions or servicing distinct user requests. The consistency of attention within halves of the graph suggests that the model has learned to identify these alternating phases of computation, where each phase involves accessing distinct, localized memory regions. This periodic shift in attention indicates that memory access patterns in Zeus are segmented into distinguishable blocks, each associated with a specific workload or phase of execution, with PARROT effectively capturing these transitions to inform its eviction policy.

We also analyzed attention layers for a single benchmark with different cache sizes. The very small, normal and very large configurations correspond to 32KB, 2MB and 8MB respectively. We conducted this study for astar, omntepp and mcf benchmarks but in this section we present only the results from astar as it captures the important insights we gained from this exercise.

For the small configuration, the attention is concentrated on more recent accesses, indicating that the model is more sensitive to temporal locality in this setup. This could imply an increased likelihood of cache conflict misses, as the cache prioritizes keeping recently accessed data. In contrast, the normal and large configurations display attention over a periodic interval of PCs, hinting at a regularity in the access pattern. This periodic attention may suggest that astar exhibits predictable, repeated accesses over specific cycles, which might align with systematic cache usage and eviction decisions. These results show that cache size is also a factor that would determine the performance of the model for a given trace because it would affect the cache state, thus affecting the Markov process and behavior of the model.

C. LLM Fine-tuning and Experiments

1) `Finetuned-Llama`*:* Having fine-tuned the 3-billion-parameter Llama model, we started assessing its effectiveness in giving feature improvement advice. To that end, we designed an experiment where we evaluated the fine-tuned model in various stages of the fine-tuning progress. More specifically, we evaluated the model at 0% fine-tuning (no fine-tuning at all), 20% fine-tuning,

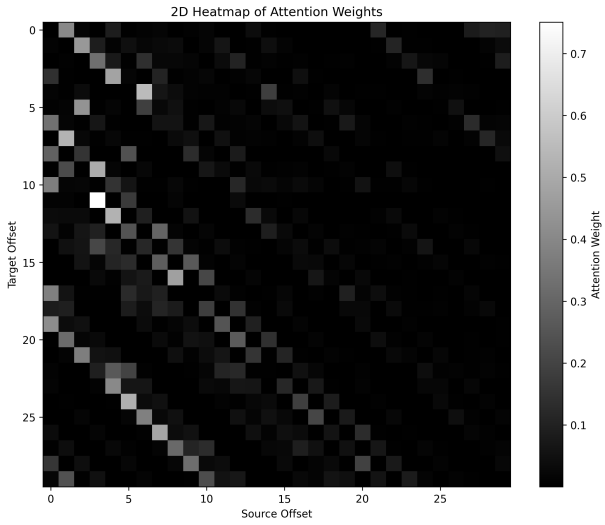

Fig. 33: Very Small

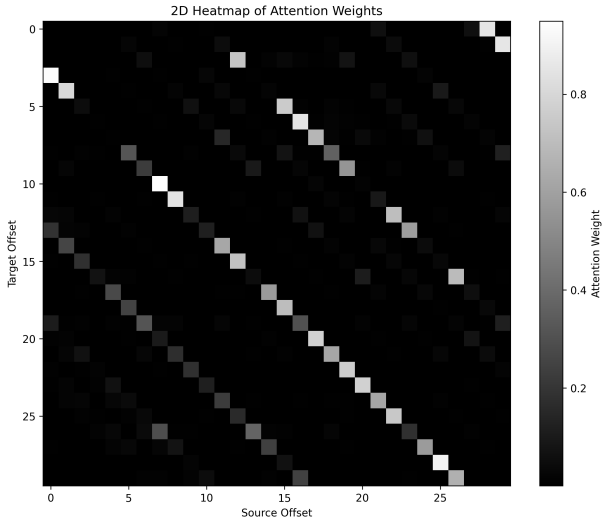

Fig. 34: Normal

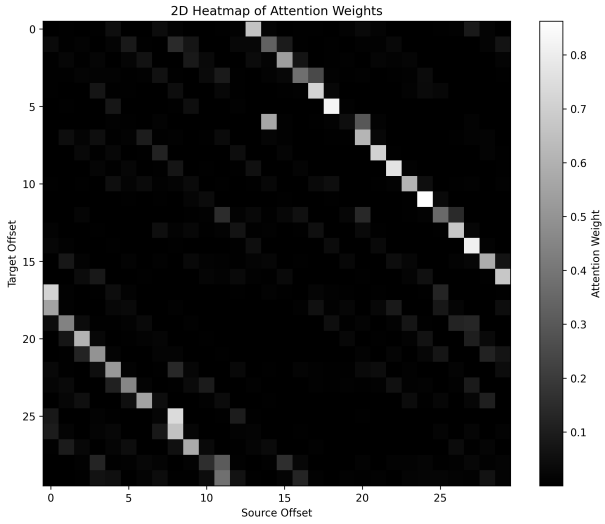

Fig. 35: Very Large

Fig. 36: astar for different cache sizes

40% fine-tuning and so forth. Each checkpoint was fed with the same prompt:

> **System**: You are a helpful assistant helping with CPU cache line eviction.
>
> **User**: This is our current prompt to an eviction policy. What can we changes should be made to help the model better identify the cache line to evict and ultimately make the eviction policy more efficient? Prompt: Current PC is <pc> Current address: <list-of-addresses> Cache lines are: <list-of-cache-lines> Eviction:

To ensure an unbiased evaluation of the responses, we employed a third-party large-language model, GPT4, to rank the outputs. This third-party model assessed each response based on its informativeness and relevance, determining which answers were the most useful and which were least effective, giving us an ordering among the checkpoints.

Our analysis[1] revealed that fine-tuning improved the quality of the model's responses, starting at being very general in nature (0% checkpoint) to being more specific (100% checkpoint).

For instance, the non-fine-tuned checkpoint provided very broad suggestions, such as leveraging features already utilized by the least recently used (LRU) eviction policy or employing hybrid eviction strategies. Although these recommendations are valid, they lack the depth and specificity we were seeking.

In contrast, the fully fine-tuned model offered a more sophisticated response, proposing the use of binary search tree representations for cache lines, incorporating cache capacity and cache line sized into the feature vectors or emphasizing the importance of analysis of frequently occurring access patterns within the traces.

2) `RAG:` We have implemented our system in a manner that readily allows it to reference multiple documents (sources of truth) simultaneously. To test the performance of our RAG ystem, we devised a set of questions designed to evaluate whether our RAG can successfully retrieve the necessary data and report it back accurately.

Our sources of truth consist of three documents:

- `raggen_1000.txt`: A file containing cache access and eviction decisions for various policies, including Belady and LRU, sourced from Homework 2.
- `arch.pdf`: The PDF version of the renowned book *Computer Architecture: A Quantitative Approach*.
- `astar_313B_test_features.csv`: A comma-separated file with features extracted from the ChampSim simulator.

[1] Results are available at https://github.com/lnotspotl/c3pu/tree/main/llama

For each question, we evaluated the system's ability to retrieve relevant data from these sources. Below, we summarize the system's performance across different scenarios[2]:

Tested on raggen_1000.txt - Correct
Question: When the PC was 0x413a5b, what did LRU evict? Is it optimal according to Beladys?
Answer: According to the provided context, when the PC was 0x413a5b, LRU evicted 0xb0b2ed1db0. However, Beladys Evicted: 0x160a69b9db0 was also recorded for the same PC. So, according to Beladys, it is not optimal, as Beladys was evicted instead of LRU.

Tested on astar_313B_test_features.csv - Incorrect
Question: Count how many times the PC 0x4090c3 occurs in the data.
Answer:I cannot find any information about the PC 0x4090c3 in the provided data.

While the former example shows that our RAG system indeed can retrieve information from different documents, it cannot work with entire files, making it impossible for it to draw conclusions about whole documents. That is, however, not the case for our next system.

3) `Llama-The-Data-Scientist`: Our final tool, dubbed `Llama-The-Data-Scientist`, is an LLM system specifically designed to analyze structured tabular data, such as access traces and memory-based features. This system addresses the limitations of LLMs regarding their constrained context windows restricting their ability to process large datasets. Consequently, such limitations also affect RAG systems, rendering them unsuitable for large-scale data analysis.

`Llama-The-Data-Scientist` functions by accepting data in the form of `pandas DataFrames`. Based on a query, it generates executable Python code, ran on the underlying dataset to produce the desired insights. This approach enables users to perform complex analyses using natural language queries, significantly simplifying the process of extracting actionable insights from data.

We hypothesize that by fine-tuning the LLM on various types of traces, we could further enhance its capability to uncover hidden correlations among features, quantities, and values, thus unlocking deeper insights from structured data and allowing us to engineer better features.

Tested on astar_313B_test_features.csv - Correct
Question: Count how many times the PC 0x4090c3 occurs in the data.
Answer:12454

[2]More example can be found here: https://github.com/lnotspotl/c3pu/tree/main/llama

This demonstrates that unlike our `RAG` system, `Llama-The-Data-Scientist` can work with entire databases, giving us the correct answer in the example above. `Llama-The-Data-Scientist` can do even more than that, for instance, generate plots and charts, as demonstrated next.

Tested on astar_313B_test_features.csv - Produces correct plot
Question: Can you plot a histogram for recency?
Answer: ...

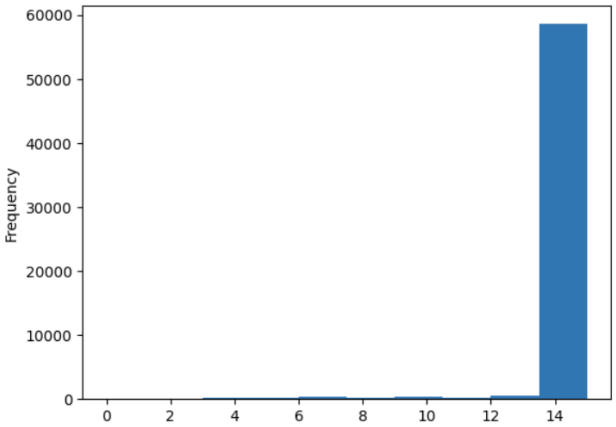

Fig. 37: Histogram produced by `Llama-The-Data-Scientist`

VIII. Conclusion

This project was investigative in nature. We extended our experiments from previous homeworks to this project for more benchmarks, identified relevant input features and evaluated them for the ASTAR benchmark, generated attention matrices for the RAG system along with some traces and inputs for the high level semantics, and finally developed an AI assistant we named C3PU that could perform data analysis of the large quantity of data in the RAG.

This work could be extended to include more features evaluated on more workloads to find conclusive results. Additionally, these could be further fed to the LLM for further fine-tuning to provide it more context and make it better at higher level data analysis.

References

[1] S. Bird, A. Phansalkar, L. K. John, A. E. Mericas, and R. Indukuru, "Performance characterization of spec cpu benchmarks on intel's core microarchitecture based processor," 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:14600331

[2] 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.

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

[4] ——, "An imitation learning approach for cache replacement," 2020. [Online]. Available: https://arxiv.org/abs/2006.16239
[5] 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.
[6] 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
[7] H. J. Yoo, J. H. Kim, and T. H. Han, "Rl-based cache replacement: A modern interpretation of belady's algorithm with bypass mechanism and access type analysis," *IEEE Access*, 2023.

APPENDIX

CONTRIBUTIONS

A.J. Beckmann

- Helped organize team meetings, delegate tasks, and contribute ideas that helped formulate our project goals and our process to achieve them.
- Helped elicit team motivation and continual progress on the project.
- Worked closely with Kaushal on the literature review for feature engineering. Helped decide which features to test and hypothesized on the best approach for applying these features.
- Modified the PARROT environment/codebase to add additional features that Kaushal provided to be used in the model.
- Took initiative to train on the features with four different feature embedding types to analyze this effect on the model performance. This was not initially in our plan.
- Extended our HW2 results by retraining the four models on five workloads with the new XOR data.
- Refactored code for HW2 tasks 2,3, and 4 for easy use on the GitHub repo.
- Achieved quantifiable results for feature engineering, formatted and analyzed its data with Kaushal, and generated corresponding graphs.
- Formatted XOR data results and corresponding graph.
- Wrote parts of the feature engineering subsections of the design, methodology, and results sections of the report.
- Helped organize and proof read the report.

Anjolie Baccay

I worked on the following tasks during this project:

- Refined subsections in Background and Related Works that originated from Homeworks 1 and 2
- Collaborated with teammates to write the final report.
- Wrote Background and Related Works sections regarding the benchmarks, benchmark overview and benchmark discussion
- Helped on formatting parts of the report
- Added some of the diagrams and corresponding discussions to the Methodology section
- Proofread the report

Avit Rane

- Wrote the "Prior works related to Cache Replacement using Machine Learning" section and all sections related to attention layer analysis in the report
- Made the diagrams in the above mentioned sections
- Wrote code to capture attention data and plot it
- Ran evaluation for 19 benchmarks
- Wrote a script to generate JSON files from the attention data which could be used in RAG
- Proof reading of report
- Helped in the final editing of report
- Participated in group discussions
- Gave suggestions to teammates on how to present the results
- Helped maintain a working conda environment which was useful towards the end of the project

Atharv Oak

- Worked with the team to discuss possible routes on action in the project
- Participated in the group discussions
- Worked on understanding the working of the RAG system and various techniques used in it
- Researched models that can be fine-tuned easily as compared to others
- Looked up 3rd party systems that would make the process of fine-tuning easier.
- Looked into various methods of fine-tuning such as SFT, Full Fine-Tuning, LoRA, QLoRA, GLoRA, etc.
- Modified the code created by Jakub so that the input files could be JSON and CSV.
- Worked in writing the LLM sections of the report
- Proof Read the report
- Worked on organizing the report better

Jakub Jon

- Proof-read this report
- Worked with the team, actively discussed ideas
- Wrote motivation and summarized results for `LLM Fine-tuning and Experiments`
- Analyzed the problem of fine-tuning and model inference
- Designed, implemented and finally fine-tuned all our LLM systems
- Together with Kaushal performed experiments using the fine-tuned LLMs
- Proudly made all our code open-source

Kaushal Mhapsekar

- Lead the team and project.

- Managed meetings, discussion forums, platforms, people, tasks, initiative and issues.
- Primarily worked on Feature Engineering tasks with AJ.
- Performed literature review for Feature Engineering.
- Modified the code bases and wrote additional scripts to extract features.
- Discussed and concluded results/trends for features with AJ.
- Analyzed the features and traces to formulate benchmark questions.
- Worked with Jakub to experiment fine-tuned LLMs for the extracted features to get insights.
- Ideated with Nishant for High-level semantic Analysis
- Collaborated with teammates to make the presentation and write the report sections - Abstract, Introduction, Feature Engineering (Background, Motivation, Design, Methodology), Conclusions, Formatting, and aided in other parts when needed.
- Coined the name **C3PU!**

Naveen G

- Trained Parrot for extended CPU benchmarks.
- Worked with teammates to write the final report and presentation
- Proofread the report.

Nishant Raman

- Primarily worked on the High-level semantic analysis parts, including report, presentation, ideation and literature review.
- Collaborated with people beyond the team, Joshua and Ethan, to form high level semantics for the class.
- Verified Kaushal's Features.py code
- Collaborated with the team and actively discussed ideas.
- Worked with teammates to write the final report and presentation.
- Proofread the report.

MEETINGS:

Our meeting logs are shown in TABLE IV

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.

RESOURCES

You can find the links to our works here:

- GitHub - https://github.com/lnotspotl/c3pu
- OverLeaf - https://www.overleaf.com/read/scrkcxzsntxd5ef26b
- Results Spreadsheet - Google Sheet

Date	Time (hours)	A.J.	Anjolie	Atharv	Avit	Jakub	Kaushal	Nishant	Naveen
11/13	1	Present	Present	Present	Present	Present	Present	Present	Present
11/15	2.5	Out of town	Absent	Present	Present	Present	Present	Present	Absent
11/22	5	Present	Absent	Sick	Absent	Absent	Present	Absent	Absent
11/25	3	Present	Absent	Present	Absent	Present	Present	Present	Absent
12/04	3	Present	Absent	Absent	Absent	Absent	Present	Present	Absent
12/12	6	Present	Present	Present	Present	Present	Present	Present	Present

TABLE IV: Meeting Log