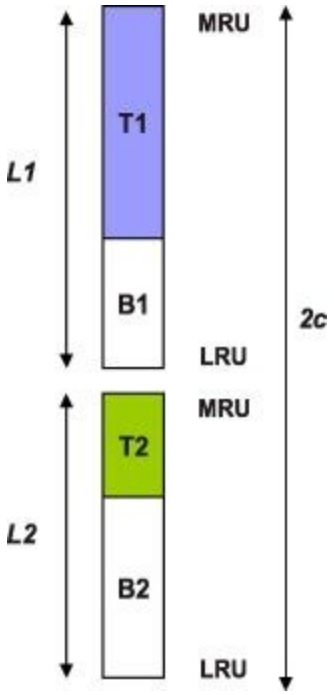
CS F407 - Artificial Intelligence Assignment
Phase 1 - Design Document

This document contains a brief overview of the original work and two techniques which aim at improving the basic, static ARC cache replacement scheme.
We intend to read more literature and implement/ improvise/ compare with state-of-the-art schemes being used currently.

1. Adaptive Replacement Cache (ARC) -

ARC improves the basic LRU algorithm by maintaining two lists, T1 and T2, for recently and frequently referenced entries. Each of these is extended with a 'ghost' list, B1 and B2, which keeps track of the history of recently evicted cache entries. The algorithm uses ghost hits to adapt to recent changes in resource usage by the running application. Thus the number of blocks which are 'allocated' for each of the two strategies keeps changing dynamically according to the workload. If LRU is more efficient, more blocks in the cache will be dedicated to storing these entries. A similar pattern is followed for LFU.



2. Driving Cache Replacement with ML-based LeCaR

Overview: Memory accesses are specific to the workload or application being executed currently. A system that can learn the specific memory access pattern of the current application and maintain and evict the cache entries accordingly would improve the speed by an appreciable factor.
An appropriate model of machine learning for cache replacement is that of Reinforcement online learning(RL), which requires that decisions be made online as requests arrive and that cumulative regret (or reward) be optimized.

Data: The data used to test the algorithm was synthetically generated to alternate between favoring LRU and LFU. It was then analyzed if the algorithm is able to learn this switch and optimize accordingly.

Technique: LeCaR assumes that at any instant, the workload is best handled by a probability distribution of only two fundamental policies - LRU and LFU. The weight associated with the probability of these two policies is not a function of their current hit rate, but of their current associated regret. In order to manage regret, the cache implements a FIFO queue of the most recent evictions from the cache. Each history is labeled with the policy that evicted it from the cache. A decision is considered “poor” if a request causes a miss and if the requested page is found in history. The intent is that a miss found in history could have been rectified by a more judicious eviction, and hence the regret. Regret is graded and is larger if it entered history more recently. When poor decisions are caused by a specific policy, that policy is penalized by increasing the “regret” associated with it. Hence, we observe that our aim at all times would be to minimize the regret with online learning as more and more data comes in.

3. LSTMs for learning patterns in memory accesses

Overview: Memory accesses can be treated as time-series data. Under the assumption that there is contextual significance to memory access, RNNs are used to model the patterns. To improve memory access time, prefetching is done based on the model’s prediction.

Data Collection: The data used is a dynamic trace that contains the sequence of memory addresses that an application computes. This trace is captured by using a dynamic instrumentation tool, Pin, that attaches to the process and emits a

”PC, Virtual Address” tuple into a file every time the instrumented application accesses memory.
This raw access trace mostly contains accesses that hit in the cache (such as stack accesses, which are present in the data cache). Since we are focused on predicting cache misses, we obtain the sequence of cache misses by simulating this trace through a simple cache simulator that emulates an Intel Broadwell microprocessor.
We use the memory-intensive applications of SPEC CPU2006. This is a standard benchmark suite that is used pervasively to evaluate the performance of computer systems.

Data Pre-Processing: Due to dynamic side-effects such as address space layout randomization (ASLR), different runs of the same program will lead to different raw address accesses. However, given a layout, the program will behave in a consistent manner. Therefore, one potential strategy is to predict deltas,

$$\Delta_N = Addr_{N+1} - Addr_N$$

instead of addresses directly. These will remain consistent across program executions and come with the benefit that the number of uniquely occurring deltas is often orders of magnitude smaller than uniquely occurring addresses. So each memory access is represented as a tuple.

$$(\Delta_N, PC_N)$$

Evaluation Metrics:

A. **Precision** - Precision-at-10, which makes the assumption that each model is allowed to make 10 predictions at a time. The model predictions are deemed correct if the true delta is within the set of deltas given by the top-10 predictions. A label that is outside of the output vocabulary of the model is automatically deemed to be a failure.
B. **Recall** - Recall-at-10. Each time the model makes predictions, we record this set of 10 deltas. In the end, we measure the recall as the cardinality of the set of predicted deltas over the entire set seen at test-time. This measures the ability of the prefetcher to make diverse predictions but does not give any weight to the relative frequency of the different deltas.