CPU Cache Simulation Library

Specifications

Background

This library is intended to simulate a *cache memory* module. Simply put, a cache is a temporary storage of data which enables faster access to that data than its original storage location. This original storage location is called a *backing store.* Data is represented as key-data pairs. The cache and backing store use the key as a unique identifier for a particular piece of data.

The cache acts as an intermediary between the agent requesting the data (e.g. a user, some other piece of software) and the backing store. The agent never directly accesses the backing store and instead only communicates with the cache. If the cache contains the data requested by the agent, then it immediately returns that data to the agent and the cache does not access the backing store. This is called a *cache hit.* If the requested data is not contained in the cache, then the cache accesses the backing store to obtain the data, saves the data within itself for future requests, and then returns the data to the agent. A *cache miss occurs* when data is requested but it is not contained in the cache.

A cache has a much smaller capacity than its backing store and thus can only hold a small portion of the overall available data. When a cache miss occurs, the cache must access the backing store to retrieve the requested data and then *installs* (i.e. saves) the data into the cache for future requests. However, if the cache is already holding its maximal amount of data, the cache must first *evict* an existing piece of data to make room for the incoming data to be installed. Notice that a cache miss can occur even if the cache is not full. For example, at the beginning of a computation the cache is initially empty and contains no data. Therefore, the *first* time any particular piece of data is requested is typically a cache miss. See also Figure 1.

An agent may also want to update (i.e. write) data associated with a particular key. This simulated cache follows a so-called “write allocate” policy: a key-data pair must be installed in the cache for data to be modified. Therefore, a write request first behaves as a read request. If the requested key is not currently installed in the cache, a cache miss occurs and the cache retrieves and installs the data from the backing store. Then, the cache modifies the installed copy of data and modifies the data in the backing store.

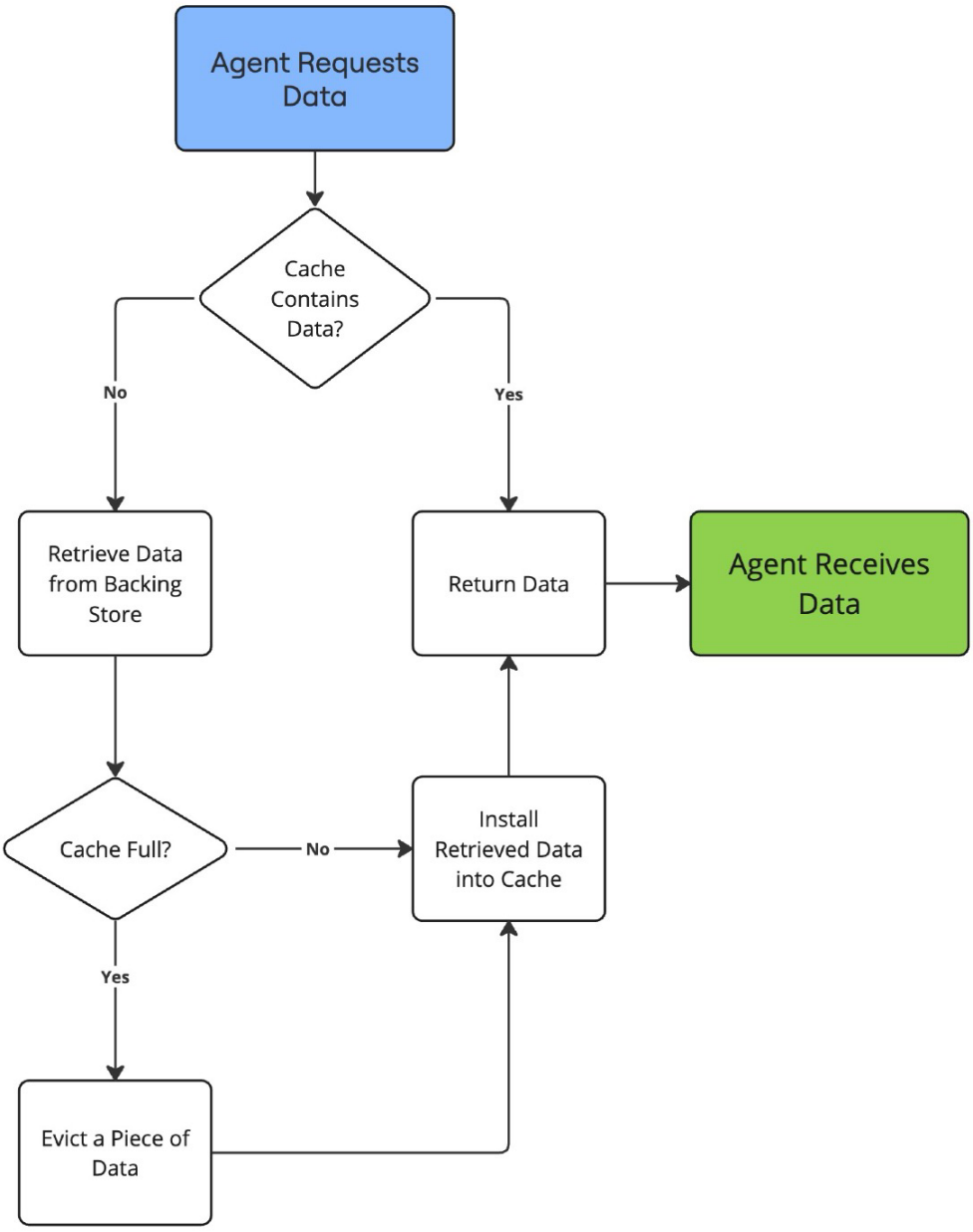

Figure 1: Flowchart of cache behaviour.

Goal

The goal of this library is to simulate a cache and measure metrics like the number of cache misses experienced during a sequence of accesses.

Class Summary

- `CacheItem`: objects stored in the cache.
- `CacheResponse`: an object storing a `CacheItem` and related metrics returned to the agent when requesting a piece of data.
- `Cache`: A data structure storing `CacheItems` and fulfilling the role of a cache
- `CacheSim`: A class to orchestrate the simulation and act as the requesting agent for the cache.
- `NotFoundException`: An exception thrown when requested data is not found in the cache or backing store.

CacheItem

The `CacheItem` class represents a single item to be stored in the cache. It consists of two crucial integer variables: key and data. Data is the actual data looking to be retrieved from the backing store and returned to the agent. The key acts as the unique identifier for the `CacheItem` and its data. Thus, the key is used by the agent to request a particular data item and is used by the `Cache` to find the requested data within itself or the backing store.

Cache Response

The `CacheResponse` class is a simple data structure returned to an agent who requests data from the `Cache`. It bundles the requested `CacheItem` alongside metrics like whether this request triggered a cache miss or not.

Cache

The `Cache` class represents a cache. It has a particular capacity to store previously requested data as `CacheItems`. The backing store is modeled as a text file and the `Cache` mimics accessing the backing store by reading the text file. The text file has a simple format of two integers per line. The integers are separated by a space where the first integer is the key and the second integer is the data.

The `Cache` follows a simple “least recently used” (LRU) replacement policy. When data is to be installed in the `Cache`, if the `Cache` is not full, it installs the data in the location with

smallest index. If the `Cache` is full, it evicts the `CacheItem` which was accessed furthest in the past and then installs the new `CacheItem` in that place.

For an agent to write data, the `Cache` first behaves as if the data was requested, installing and evicting data, as needed. But, before returned the requested data to the agent, the Cache first modifies the installed `CacheItem`'s data and writes the new data to the backing store.

If an agent ever requests a piece of data which is not in the `Cache` and not in the backing store, a `NotFoundException` is thrown.

CacheSim

The `CacheSim` class orchestrates a sequence of accesses to a `Cache` object and records the history and effects of those accesses. Given a sequence of keys to access via the simulate() method, `CacheSim` makes a sequence of requests to the Cache and records the `CacheResponse` returned from each request. `CacheSim` also records the contents of the `Cache` before and after each request. The contents of the `Cache` are recorded as a `String` and thus the history is a sequence of `Strings`. `CacheSim` provides a helper method to convert the current contents of the underlying `Cache` to a `String`.