N-way Set Associative Cache library design.

Subject. The subject of this work is to design the software library that provides the caching functionality according to the following requirement:

1. The cache itself is entirely in memory (i.e. it does not communicate with a backing store).
2. The client interface should be type-safe for keys and values and allow for both keys and values to be of an arbitrary type. For a given instance of cache all keys and values must be of the same type.
3. Design the interface as library to be distributed to clients. Assume that the client doesn't have access to the source code.
4. Provide LRU and MRU replacement (eviction) algorithm.
5. Provide a way for any alternative replacement algorithm to be implemented by the client and used by the cache.

Interpretation of requirements.

1. Since we are limited to using the main memory for the cache location, we must consider the physical limitations of the environment this library is being used in. My solution uses two array objects: one for negative values of hash keys and one for positive values of hash keys. This allows the cache address space division into Offset, Set Address, and Tag Address (see Theory section below). At the same time, it eases the restriction of the continuous memory region that needs to be allocated to one large array object. Also, the provided library implementation restricts the total number of cache entries by 2^{25} = 33,554,432. It's easy to lift that restriction and go up to 2^{31} = 2,147,483,648 that is a limit imposed by the System.Int32 data type.
2. This requirement dictates using generic types for both Keys and Values. Both Cache<TValue, TKey> and other supporting types (Config<TValue, TKey>, CacheItem<TValue, TKey>, etc.) have been designed according to this requirement.
3. This requirement assumes documenting the source code as well as providing a programmers reference/samples document.
4. Implemented as required.
5. Implemented as required (see Replacement/Eviction Strategy notes below).

Theory. The need to have a cache is obvious since the amount of data applications operate on always exceeds the resources (RAM and CPU cycles) modern information systems have at their disposal. There are three main types of caching algorithms that are most widely used:

- Directly Mapped Associative cache;

SET ADDRESS	OFFSET ADDRESS

Directly Mapped eliminates the need for the cache item scans but it is done at the expense of the increased number of collisions/evictions. Every time a key's hash falls into the set address space and if that slot is occupied then eviction happens and we must deep in to the main (potentially slow) storage for the value to bring it into the cache. The good hashing algorithm that produces random integer values uniformly across the whole range between int.MinValue and int.MaxValue will minimize the number of collisions. The collisions may be eliminated entirely if number of available addresses in the cache equals or is greater than the number of items in the main storage.

- Fully Associative cache;

TAG ADDRESS	OFFSET ADDRESS

Fully Associative cache minimizes chances of collisions and evictions since set separation boundaries are removed in cache address space is filled in as uniformly as key hashing algorithm allows. The down side of the Fully Associative cache is the scans in search of cached items become lengthy.

- N-way Set Associative cache;

TAG ADDRESS	SET ADDRESS	OFFSET ADDRESS

This is a hybrid of the above two cache types. It provides the compromise between the number of evictions and the duration of cache item scans. There is always an optimal combination of cache parameters for a given data set and hardware/infrastructure and the user of this library is in the best position to find that combination. This brings us to the topic of Library Controls (see below).

NOTE: I limit the OFFSET address bits to one bit in my library. It would be a good future improvement for the library if we allowed users to control the number of offset bits.

Controls.

- **Set and cache sizes** are controlled by the number of bits used by SET address and by the TAG. I decided to use the number if bits as opposed to the size and number of items per set to underscore the importance of the proper memory alignment of the sets inside of the cache

address space. This way both cache size and set size are the multiples of 2 and cache address space is not wasted.

- **Replacement/Eviction strategy.** My library implementation provides standard LRU (default) and MRU algorithms. There is also a way to hook up the custom `ReplacementStrategyHandler<TValue, TKey>` delegate instance into the library that provides flexible interface and ease of use.
- **Loading initial cache items.** A cache may be given a reference to the custom implementation of the `ICacheLoadProvider <TValue, TKey>` interface to cause loading of the initial cache data.
- **Number of scan threads**. This library may potentially use multiple scanning threads in case Set size is big enough. Library user may control the maximum number of scan threads used.
- **Offset and continuous memory requirements.** As mentioned before, I limit the OFFSET address bits to one bit in my library which results in splitting the cache into two physical memory arrays. It would be a good future improvement for the library if we allowed users to control the number of offset bits.
- **Memory footprint**. The main control for the memory footprint that users of the library have is the number of bits that are used to form the complete cache address. The total memory consumption is also a function of the VALUE and KEY type used.
  TOTAL MEMORY USAGE >= $2^{(\text{address bits})}$ * (80 + (VALUE length) + (KEY length))

- **Key Hashing Algorithm.** It is VERY important to employ a hashing algorithm for the `TKey` type that is fast and yet of high quality meaning that it produces random integer values uniformly across the whole range between int.MinValue and int.MaxValue. In my test project, I use System.Int32 type for the keys which is ideal. Still, library users may use whatever key types they wish and can pass in the custom implementation of the `Key2HashHandler<TKey>` delegate to control hashing function.
- **Cache expiration**. The library provides the control of the cache expiration interval.
- **Cache Push/Pull scenarios**. Users may completely get away from making Push calls by simply providing the custom implementation of the `ValueLoadHandler<TValue, TKey>` delegate. If this done then that implementation will be called every time an item with given key is not found in the cache or it's expired. In case this implementation is not provided then the library client is responsible for analyzing the returned item's timestamp, making calls to the storage and pushing loaded item into the cache using the Push method. Cache library will automatically discard expired cache items ONLY if the custom implementation of the `ValueLoadHandler<TValue, TKey>` delegate is provided.

Samples and documentation. This library source code has XML comments embedded in it, so the help authoring tools like Sand Castle SHFB can be later used to generate programmer's reference. This library

solution comes with the test project that may be freely distributed to the customer and used as a use case example. The Figure 1 below is the screenshot of the test project form.

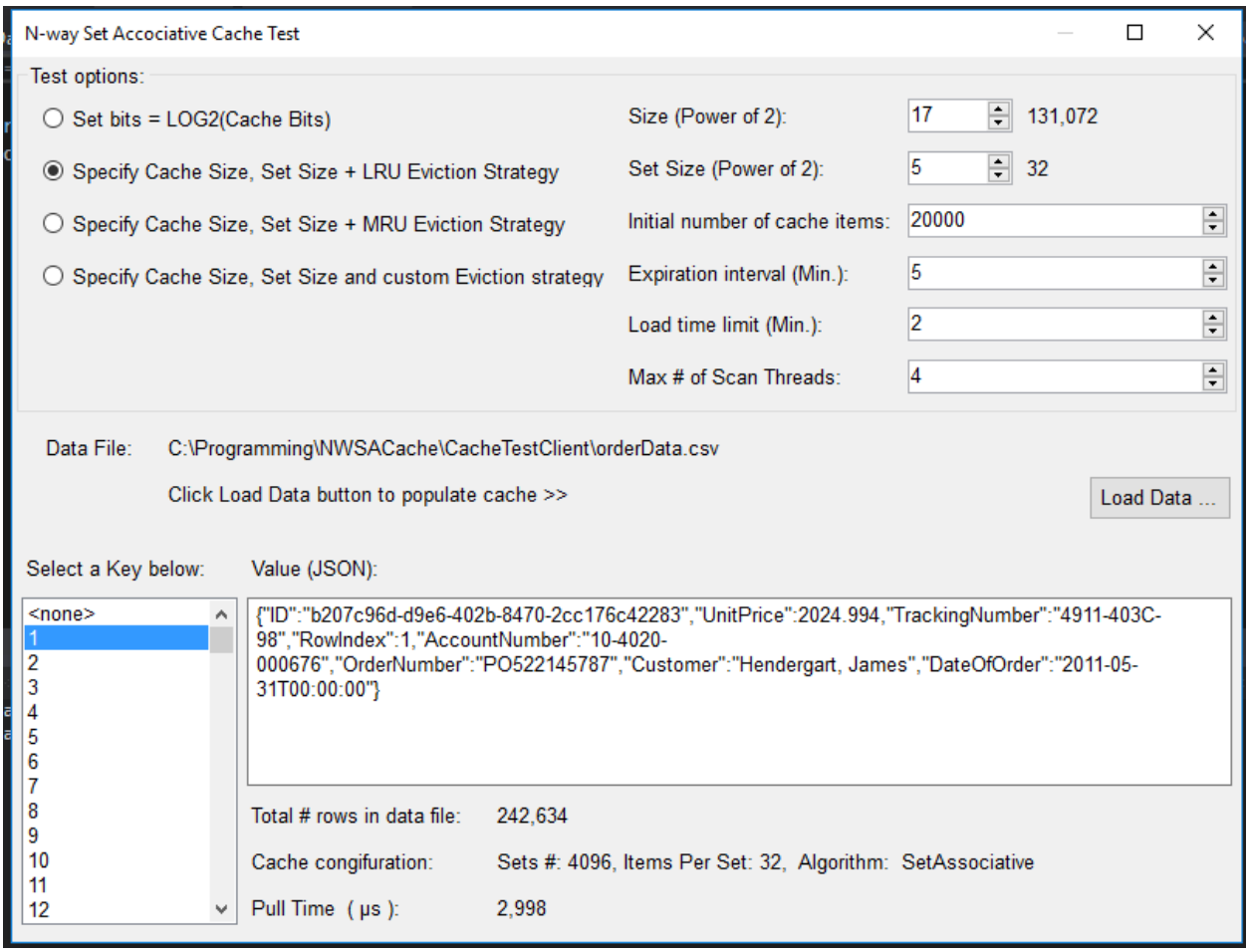

FIGURE 1

As can be seen, this test project form allows using the following controls to make the cache function be tested in various configuration parameter combinations:

- Cache and set sizes;
- Replacement/Eviction strategy including custom eviction strategy;
- Initial number of items loaded into the cache from the backing store;
- Max load time of the initial cache;
- Cache expiration interval;
- Max number of the cache scan threads;

Test data and test object. The test data comes in a form of CacheTestClient\orderData.csv file and contains 242,634 rows of data. Test `OrderDetail` class defined in CacheTestClient project is used as a cache value type. `OrderDetail` class overrides ToString() to return its JSON representation example of which may be observed at the Figure 1 above.

December 2, 2017

Vladimir Pereskokov