Dictionary: Best practices

May 27, 2018 forwolk Uncategorized 0

Every developer uses C# Dictionaries on a daily basis. Their perks are obvious: easy to use API, constant time O(1) lookups, addition, and removal operations. Still, there are some cases in which misusing Dictionaries can lead to mediocre performance or degradation of the lookup operation speed to O(n). I am going to cover those popular pitfalls in this post.

Dictionary internal design

Under the hood, Dictionary is operating in the following way. Similar to List<T>, Dictionary has an underlying array which stores its elements. It is called “bucket array”. Whenever we try to add an item to a Dictionary, a GetHashCode() operation is executed on the key being inserted. Then a compression function is applied to the hash code to produce an index inside bucket array where we are going to store the element.

Compression function in C# Dictionary is pretty straightforward:

hashCode % bucketArray.length

As the size of our bucket array is not infinite, compression function application may often result in “hash collisions”, a situation when objects with different hash codes end up being mapped to the same index. For example, if the size of our bucket array equals 10 and we are adding two items with hash codes equal to 20 and 30, they will both map to a bucket with an index 0 as in both cases the result of the mod operation will be 0.

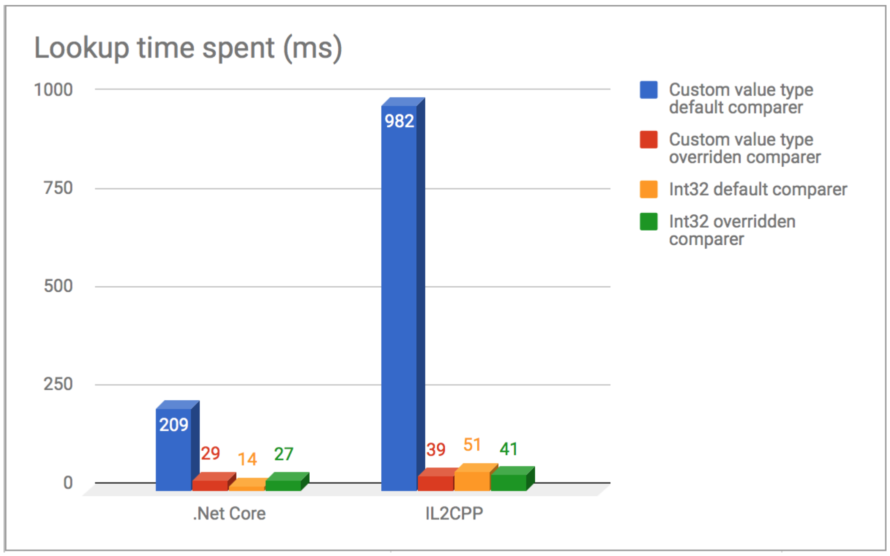
Storing two different items in one slot is impossible, that’s why we have collision resolution strategies. C# Dictionary uses so-called “chaining”. It means that every bucket is actually represented with a linked list kind of a structure. In case several keys collide they are all getting added as elements of the linked list. When you retrieve an item from the Dictionary, it will first calculate the bucket index of your key as mentioned above. Then it will iterate over the linked list in that bucket (which may have a size of 1) searching for the same element. You can check out the linked list part of Dictionary implementation here.

Now that we are through with internal workings of the Dictionary let’s move on to mistakes developers usually make.

Common mistakes

1. Forgetting to supply custom IEqualityComparer<T> when using Value type as a key in Dictionary.

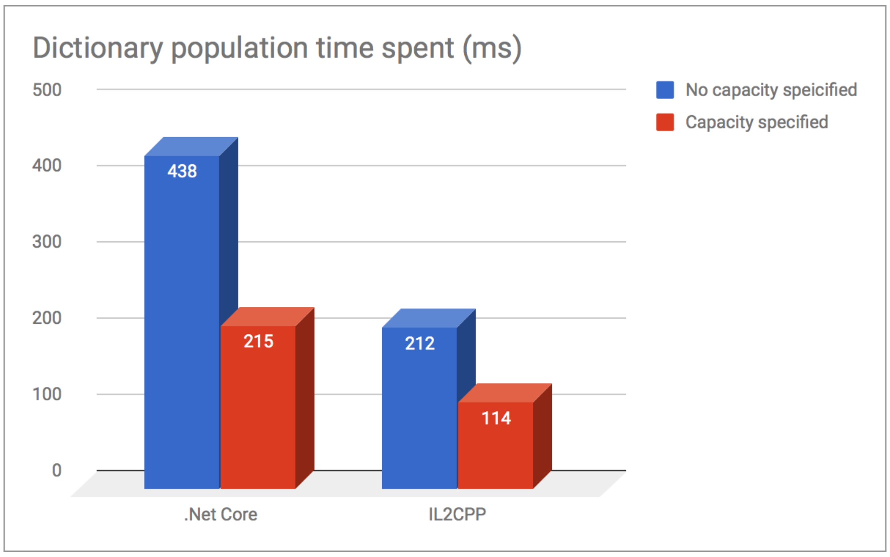
This leads to boxing of the keys every time we perform any operation on the collection. Even lookups will be hurt by it. If you do not supply your implementation of IEqualityComparer<T>, C# dictionary will use the default one.



As you can see, the difference is obvious. Though it seems that .net core compiler or JIT are doing something magical with default integer comparators as they actually perform better than the overridden once. Still, the statement holds for IL2CPP output, and I suspect for older versions of .net.

2. Omitting initial capacity of the Dictionary when you know its maximum size in advance.

Omitting capacity will make Dictionary rehash and reposition items in their respective buckets every time it's underlying bucket array needs to grow.

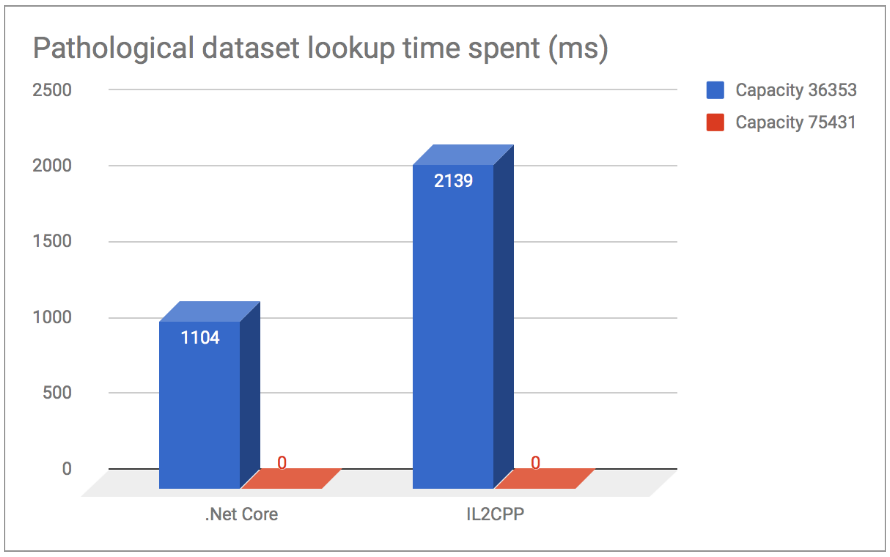


3. The final but not obvious case when Dictionary behaves far from optimal is when you supply it with a pathological data set.

Pathological data set means that a lot of the items resolve to the same bucket, thus reducing lookup operation to O(n). The most straightforward example of the pathological data set is a custom class whose every instance returns the same hash code. All of its instances will map to the same bucket. To describe another case, we will have to consider how Dictionary "grows".

We already mentioned that collisions are the result of the compression function which is **hashcode % bucketArray.length**. As you can see elements, for whose hashcode, bucketArray.length is a factor, will all map to 0 index bucket. To reduce the probability of that happening, Dictionary grows its underlying array in a different manner than List<T>. Instead of just doubling the array, it doubles the current capacity and searches for a prime number closest to that value – this will be new array size.

Prime numbers are rare, so the probability of them being a factor of your keys hashcode is relatively low. Still, under certain "size" of underlying bucketArray and a certain set of data, there is a possibility for this case manifesting. Here is an example.



The trick here is that all elements have hashcode equal to 36353 * n. So in case of Dictionary capacity being equal to 36353, they all map to the same bucket.

So, the important takeaways should be:
1. Always override comparer in case you are using value types as keys.
2. Specify capacity of the Dictionary if it's known, prior to populating it with data.
3. Avoid pathological data sets to keep Dictionary's performance close to O(1).

Here is the link to a benchmarks code.

P.S.
I would like to thank my good friends Maxim Agapov and Pavel Naidenov for reviewing this post.

NEXT »
IL2CPP optimisations