ICU UnifiedCache Design

by Travis Keep

31 July 2015

Goals

- To provide a single, bounded cache for all of ICU.

Non Goals

- To prevent client programs from leaking memory. In particular there is no bound on the number of objects in the cache that the client directly references (This does not increase memory footprint since the client already is holding onto those same objects). The only bound is on unreferenced objects kept around to make cache retrieval of them faster when they are finally needed.

High Level Design

The UnifiedCache is a singleton cache that is a UHashtable where the keys are polymorphic and the values are pointers to shared objects. These shared objects allow multiple keys to be mapped to the same value and allows the cache to share its cached data via const pointers rather than provide each client with its own copy of the cached data.

Clients signal that they are done with a value in the cache by calling removeRef() on it (all values in the cache extend SharedObject which provides reference counting functions). When a cached value has no more client references, it becomes eligible for eviction from the cache.

The eviction policy of the cache is random instead of systematic like least-recently-used. However, eviction never happens unless the cache statistics reach some minimum threshold. The statistic we care about is the number of entries in the cache that could be removed without removing a value that a client is using. We call this the unused count. Eviction does not happen unless the unused count exceeds the maximum of some fixed value and some percentage of the entries in use count (the entries that cannot be removed without removing a value that a client is using). Clients of the cache are free to set these thresholds; however, we do provide a reasonable default that we believe is suitable for most applications. Note that (unused count) + (in use count) = (total cache key count) and (in use count) == (number of unique values in cache)

To be clear, we bound only the unused count. We do not try to guess how much memory each entry uses nor do we weigh entries based on entry value type.

When unused count does reach the minimum threshold, eviction is triggered whenever events happen that would increase the unused count such as when the last client releases its reference on a shared value. When such an event triggers eviction, the cache runs a small slice of fixed size of the eviction job. This way, the full cost of eviction is amortized over all the cache operations so that the eviction cost per operation remains constant regardless of the size of the cache. Because we amortize cache eviction and because we inspect cache elements at random, we cannot guarantee that the unused count will always be below the threshold. However, we can guarantee that the cache will remain bounded as long as clients do not leak resources by failing to remove their references to unused cached values. In particular when the unused count exceeds some small percentage of total cache size(10% today), the expected number of entries evicted per eviction slice will exceed 1 (because we examine 10 entries in each slice), and each eviction slice gets triggered when unused count increases by 1. Therefore the unused count stabilizes over time.

Detailed Design

Classes

The SharedObject class

This is the class that all cached values extend. SharedObject contains the following fields:

- **totalRefCount** - An atomic int. This is the total number of references to this object, when this count reaches 0, the object is automatically deleted.
- **hardRefCount -** An atomic int. This is the total number of references to this object outside the cache. When a SharedObject is created and used apart from the cache, hardRefCount == totalRefCount (unless an addRef or removeRef is in progress)
- **softRefCount** - A regular int. This is the total number of references to this object from within the cache. Its value is precisely the number of keys in the cache that have a pointer to this object. Only cache code is allowed to modify this field. It does so only when holding the global cache mutex (discussed in the UnifiedCache section below)
- **cachePtr -** Pointer to the cache. Will be NULL if this object created apart from the cache. This is set once when the object is being initialized and then never changed. This pointer allows the SharedObject to notify the cache of whether or not at least one client is using it. This way, the cache can perform an eviction cycle at the appropriate times.

SharedObject has the following methods

- **addRef() -** totalRefCount++, hardRefCount++
- **addSoftRef() -** totalRefCount++, softRefCount++
- **removeRef() -** hardRefCount--, totalRefCount--
- **removeSoftRef()** - softRefCount--, totalRefCount--

To ensure totalRefCount >= hardRefCount + softRefCount (otherwise, the object could be deleted while in use), we always increment totalRefCount first and decrement it last.

The CacheKey<T> class

This is the base class for all the keys used in the cache. Each key knows how to create its corresponding value in case of a cache miss. The template variable T, indicates the type of the value that the key creates. Each CacheKey has a createObject() method that can either:

- Create a brand new object using new. In this case, it must add one hard reference to it using addRef()
- Fetch an existing object out of the cache using some computed derived key. This second option is how we can have multiple keys pointing to the same value.

Clients are expected to subclass CacheKey<T>. Each subclass can contain arbitrary key information. however, all subclasses must provide implementations of these operations:

- **Copy constructor** - While the cache shares values, it owns its own copy of all the keys

- **clone() -** polymorphically copies this key.
- **operator==** - Determines if two keys are semantically equivalent.
- **hashCode()** - The hashcode. If two keys are equivalent, then their hash code must be equal. The converse need not be true.
- **createObject() -** On cache miss either creates a new value or fetches an existing one from the cache

The LocaleCacheKey<T> class

Because most cached values are keyed by locale, we provide this stock implementation of CacheKey that uses a Locale object as the key material. Clients are still responsible for providing a createObject() method for each T type they cache by locale.

UnifiedCache class

This is the singleton cache. To support thread safety, it provides a single global mutex that must be locked whenever the cache is read from or written to.

The cache has the following fields

fHashtable - The UHashtable
fEvictPos - The position in the hashtable where the next eviction cycle should begin.
fItemsInUseCount - We store the number of entries in use and compute the entries not in use count using the formula fHashtable.length() - fItemsInUseCount.
fMaxUnused - This is the minimum fixed number of unused entries needed before eviction cycles begin
fMaxPercentageOfInUse - This is the minimum value of (not in use * 100 / in use) needed before eviction cycles begin

It has the following public methods:

- **get() -** Gets a shared value from the cache. The caller must call removeRef on the pointer get provides when it is done. get() is templated based on the type of cache key provided so that no awkward type casting is needed. One version of get() allows caller to provide a context object as a const void * allowing createObject() to use that context when creating objects.
- **static getByLocale()** templated convenience routine that creates the appropriate LocaleCacheKey<T> type based on the type of pointer the caller provides.

It also includes methods to allow SharedObject classes to notify the cache of whether or not they are in use

- **incrementItemsInUse** - Increments the entries in use. Cache global mutex must be held. This is safe because the cache always increments the hard reference count while holding the mutex before handing out the pointer to the first client.
- **decrementItemsInUseWithLockingAndEviction** - Decrements the entries in use and initiates an eviction cycle if thresholds are reached. Cache global mutex must not be held. This method is called when the last client calls removeRef() on a SharedObject.

Unused Count

The number that we limit is the total size of the cache minus the entries in use count. These are precisely the entries that while making lookups slightly more efficient are not essential to the operation of the cache. We call this the unused count since the client is not actually holding onto data in these entries.

Master Entries and alias entries

For each unique value in the cache, there will exist exactly one master entry. Additional keys that point to this same value comprise “Alias” entries. Here is the difference between a master and alias entry.

- When the createObject() method of a key fetches a value from the cache using a derived key, that creates an alias entry in the cache.
- When the createObject() method of a key creates a new object from scratch, that creates a master entry in the cache.

Eviction

In order to bound the unused count, it is necessary to periodically evict entries from the cache. Only entries eligible for eviction will ever be evicted.

Criteria for eviction eligibility

Entry must not be a master entry OR the entry’s value must be free of BOTH all client references (hard references) AND references from any “alias” cache entries (soft references). The value in such an entry would have 0 hard references and 1 soft reference from the sole entry containing it.

Eviction mechanism

Entries that are eviction eligible are evicted in some random order as opposed to least recently used. We choose random eviction because it both does the job AND is easy to implement. We hope it will do well enough.

We evict entries from the cache round robin style iterating through the cache looking for eligible entries. When the end of the cache is reached, we start over at the beginning. An eviction slice, which examines the next few entries (today 10) evicting if necessary is run whenever the unused count is at or above some user provided threshold AND:

- A new entry is added to the cache that is not a master entry
- The client removes all references to a value in the cache

This implies that some cache maintenance is done when client calls get() on the cache as well as when client disposes of its references to a cached object.

The purpose of running eviction slices rather than the entire eviction pipeline is to reduce contention for the cache mutex which must always be held while eviction is taking place. Although the eviction pipeline is never run all at once, we show in appendix A that running these slices is enough to bound the cache provided that the client also bounds how many references it holds to entries within the cache.

Why is this thread safe and free from races

In this cache design, we increment and decrement multiple independent atomic variables in each SharedObject operation. While operation on each atomic variable are themselves consistent, performing operations based on the values of multiple atomic variables may cause data races. In this section, we show that there are no data races on our handling of SharedObject. Below is a table of the conditions the cache code looks for as well as

- What triggers the condition check
- Whether or not the cache mutex is held during the check
- The result of the condition holds true

#	Condition	Trigger	cache mutex held?	result
1	hardRefCount up to 1 AND shared object part of cache	addRef()	YES otherwise program terminates	incrementItemsInUse()
2	hardRefCount down to 0	removeRef()	NO	decrementItemsInUseWithLockingAndEviction()
3	totalRefCount down to 0	removeRef()	NO	delete shared object

4	hardRefCount == 0 && softRefCount == 1	UnifiedCache::_isEvictable()	YES	
5	hardRefCount == 0	After createObject() returns	NO	safeguard to ensure that createObject implementations add a hard reference.
6	softRefCount == 0	Before adding entry to cache	YES	Makes entry a master

A few key observations:

1. A thread must hold the cache mutex to change softRefCount (This isn't from the above chart. It is because softRefCount is a plain int and the cache mutex must be held to either read or modify it)
2. A thread must hold the cache mutex to raise hardRefCount from 0 to 1
3. Condition #6 is consistent because of observation #1.
4. Condition #5 may not be consistent, but this is o.k because this check is only a safeguard to ensure that the developer did the right thing. The worst that could happen is that the program doesn't terminate immediately if the developer forgets to add a hard reference in createObject()
5. Condition #4 is consistent because hardRefCount can't change between the condition check and the result being fired because hardRefCount can't get smaller than 0, and since the cache mutex is held, it can't get larger than 0 either (See observation #2). softRefCount can't change either because of observation #1.
6. Condition #3 has always existed in our code. It is safe because if totalRefCount drops to 0, then no other threads can have a handle to the SharedObject being deleted
7. Condition #2 is eventually consistent because the decrement and check of hardRefCount is atomic. Even though another thread could come in and increment hardRefCount to 1 before we get a chance to fire the result, calling decrementItemsInUseWithLockingAndEviction() is always correct because we must eventually record that hardRefCount reached zero as a result of our atomic decrement and check. The calls to incrementItemsInUse() and decrementItemsInUseWithLockingAndEviction() may come out of order, but because of the atomic behavior of increment and check and decrement and check, each will be called the correct number of times guaranteeing eventual consistency. In the worst case, the number of in use entries stored in the cache may be temporarily incorrect causing a gratuitous eviction cycle, but eventually the number of in use entries will be consistent.
8. Condition #1 is eventually consistent for reasons similar to observation #7.

Appendix A: Proof that running eviction slices that examine the next 10 entries bounds the cache

We prove that if the unused count is continually larger than the threshold k, that unused count will never exceed max(k, n / 10) + n / 10 = max(k + n / 10, n / 5). where n is the total size of the cache.

The basic idea is that if k > n / 10 and there are k entries, for each unused entry added, the corresponding eviction slice is expected to remove k / (n / 10) > 1 unused entries. The size can grow up to n / 10 more if all the unused entries happen to be all grouped together at the end.

If k < n / 10 and there are n / 10 unused entries, for each unused entry added, each eviction slice is expected to remove 1 unused entry which keeps the unused entry count at an equilibrium . Again the size can grow up to n / 10 more before eviction starts if all the unused entries are grouped together at the end.

Rigorous proof

If k > n / 10:

Suppose there exactly k unused elements so that our eviction policy begins. If we add n / 10 more unused elements, we run n / 10 more eviction slices. In the worst case we visit all the elements that were in the cache getting back to where we once were in the cache evicting the previous k unused elements without seeing any of the n / 10 new unused elements we added. So we added n / 10 unused elements but were given the opportunity to remove up to k, more than n / 10, elements so our unused element count remains at k after finishing one cache iteration. Average case we see some of the unused entries we recently added while iterating over the cache which means we would need slightly more than k / 10 eviction slices to get back to where we were. Although each of these extra eviction slices needed to get back to where we started in the cache would be triggered by adding 1 additional, unused entry, each would remove 10 entries from the cache. Which means we stay at k.

In the worst case, we could add no more than n / 10 unused entries atop the already existing k unused entries before we start evicting the original k entries. Therefore the most unused entries we could ever have is k + n / 10

If k < n / 10:

Suppose we have exactly k unused elements. Using similar arguments as above, adding unused entries to invoke enough eviction slices to loop through the cache once in the worst

case adds (n / 10 - k) elements resulting in n / 10 unused elements, but since n will have grown by the number of entries added, the total number of unused entries is still less than n / 10.

Suppose we have m where k < m < n / 10 unused elements. Using the same sort of argument we can show that the number of unused elements stays < n / 10 after the eviction process goes around once.

Again in the worst case we may have to add up to n / 10 unused elements before our eviction slices start evicting so the upper bound is n / 10 + n / 10 = n / 5.

Caveat: A cache resize could happen during eviction in which case there is no guarantee that iterating over the whole cache would visit all the elements previously there. However a cache resize is a very exceptional case because as we approach the limit for unused entries, we can expect an eviction slice to remove 1 unused value for each unused value added. In the very rare case that the number of unused entries goes over the proven bound, we can show that each eviction slice is expected to remove > 1 unused entries for each unused entry added.