Proposal for an Efficient Single Object Allocator

Dhruv Matani

April 25, 2013

Contents

1 Motivation

With increasing memory sizes, and increasing difference between the size of CPU caches and main memory, improving locality and reducing the per-object allocation overhead seems to be a desirable thing.

The C++ memory allocation model allows allocators to take advantage of the fact that the type (and hence the size) of the object being allocated (and deallocated) is known told to the memory allocator run-time by the language itself. Many allocators fail to take advantage of this subtle fact.

Containers such as `list, map, set, etc...` are used very often, and result in problems such as:

1. Memory fragmentation since nodes are not necessarily freed in the same order as they are allocated. If an allocator uses free lists to string together objects of the same size, then objects lying on different cache lines and pages tend to get grouped together.

2. High per-object overhead. A linked list node for `list<double>` on a 64-bit system occupies $8 \times 3 = 24$ bytes. If the allocator overhead is 8 bytes per object, then we are looking at a 33% overhead, which is pretty high. Some allocators try to alleviate this problem by allocating small objects in *arenas or pages* and rounding off the address of the deallocated memory to the arena or page boundary to determine the metadata related to the memory being deallocated. These allocators have a constant overhead, which is not proportional to the number of objects allocated. However, they suffer from memory fragmentation[1] since free blocks are usually strung as a linked list.

C++ allocators accept an optional *hint* parameter, which is usually ignored by most allocators. If the *hint* parameter is set by containers such as *list, map, set,*

[1]By *memory fragmentation*, I mean the behaviour of the allocator to return memory from different pages even though free memory on the same page is available. Usually, *free-list* based allocators tend to return memory in the order it was deallocated.

etc...., then it could potentially increase the cache usage of the processor if the allocator allocates objects near the object at the location of the *hint* parameter.

2 Current state-of-the-practice

Note: I have borrowed the term ***state-of-the-practice*** *from Dr. Michael Bender's presentation on Tokutek*

The current `bitmap_allocator` is attractive because:

1. The per-object overhead is just 1-bit
2. The allocator metadata and the user memory are at separate locations, so updating the allocator metadata never touches memory close to the user memory. Since the bitmaps are very tightly packed together, it means that the allocator's working set is very hot and will hopefully reside in the processor cache.

The current `bitmap_allocator` achieves the objectives mentioned above except for the following drawbacks:

1. The bound on the worst-case cost of a single allocation request is $O(n)$. However, on the average, the running time is $o(\log n)$ (yes, that's a little-Oh) since the average case is assumed to be a sequence of allocation requests or that there are enough free objects available.
2. The free-list-bitmap comes just before the actual memory, so faulty programs run the risk of corrupting the allocator metadata.
3. The code seems to be unnecessarily complicated and doesn't have a standard data structure backing it when in fact a *Segment Tree* is the perfect data structure for such a use-case. The *Segment Tree* also provides an upper-bound of $\emptyset(\log n)$ on the worst-case running time of a single call to the `allocate()` function.

3 The Static Segment Tree

The Static Segment Tree[2] is a data structure commonly used in a geometric setting to do *range queries*. It is efficient at performing *range aggregation queries* over a static data set. Fortunately, that is all we need it for.

The Segment Tree is a summary data structure that stores data in a heap-like tree-ordered fashion, with the root node holding the summary information for the complete data set, the left & right children of the root node holding the summary information for the left & right half of the data set, and so on till you reach the leaf nodes, which hold information for just a single data item. It is easy to see that the space requirements for storing n elements is $2n$, which is an $O(n)$ space requirement to store n elements.

For the purposes of the `bitmap_allocator`, we shall use a segment tree of *bits*, which means that we shall use $2n$ bits to summarize n objects (or memory regions) each of which correspond to a single object returned by the `allocate()` function.

3.1 Locating free objects

A set bit (`1`) at an internal node indicates that there exists a free object somewhere below this node, whereas a reset bit (`0`) indicates that there is no free object below this node. This allows us to quickly determine whether we should go down this node in search of a free node or not.

3.2 Locating a free object close to a given allocated object

To respect the *hint* parameter that might be passed by the caller in the `allocate()` function, we locate the leaf-node that the allocated `hint` pointer points to and try to work our way *up* the tree from there, stopping at a node that has a set

[2]`https://en.wikipedia.org/wiki/Segment_tree`

(`1`) bit and working our way down from that node using the same algorithm as before.

An idea that is slightly wasteful of space has been shown here diagrammatically: `http://bit.ly/dS2lrh`

4 The Big Picture

Similar to the older `bitmap_allocator`, we use a structure that is composed of exponentially increasing sized Segment Trees. The structure is composed of Segment Trees that have sizes 16, 32, 64, 128, 256, etc… as the memory allocation requests increase.

A pointer to each such Segment Tree is stored in an array sorted by the start address of the memory region that each Segment Tree points to. In the worst case, we will have $\Theta(\log n)$ Segment Trees, so doing a linear search on them to find a Segment Tree that has a free object is acceptable. This linear search is followed by a lookup into the Segment Tree to actually find that object and mark it as allocated.

The same strategy is used to locate the Segment Tree to which an object to be deallocated belongs. i.e. We just traverse the array of Segment Trees and check if the pointer being deallocated lies in the active range of the Segment Tree.

5 Optimizations

The older `bitmap_allocator` was parameterized by *type*, and that meant that a separate memory pool was used for `bitmap_allocator<int>` & `bitmap_allocator<float>`. Instead, we can use one more level of indirection and create an implementation `bitmap_allocator_impl<SIZE>` that is parameterized on the size of the object rather than the type of the object. This allows multiple objects having a different type, but the same size to share the underlying memory pools.

Note: This optimization is currently **NOT** implemented since objects of different types tend to have different lifetimes that are related mostly to the type of the object rather than the size of the object. Additionally, we expect there to be locality of reference as far as different objects of the same type rather than size are concerned.

We need not incur a cost of $O(\log n)$ per `allocate/deallocate` request. This is because we can walk the tree from the place we last left off. For example, if we start off with an initially empty tree, we will go down $\log n$ nodes till we reach a leaf node, after which we go up just 1 step and cache the location of this node. The next time we need to allocate a node, we will just visit the right child of this node and walk up 2 steps to the grand-parent of the allocated node. This results in an algorithm that has a *worst-case* cost/operation of $O(\log n)$, but the amortized cost/operation is $O(1)$ if we ignore degenerate cases. In *expectation* though, this algorithm incurs a cost of $O(1)$ per operation (i.e. `allocate/deallocate`).

The degenerate case arises when we allocate the *last* free block in a segment tree and immediately deallocate it, and repeat this over and over again. This causes the algorithm to repeatedly reset and set $O(logn)$ bits from the root to the affected leaf node.

We notice that we need not update the cache location of our node in case of a `deallocate` operation since a `deallocate` operation only frees nodes up, which is okay as far as our optimization is concerned. The cached node location is updated only when we allocate nodes.

Another optimization we use is that we cache the most recent Segment Tree from which an allocation/deallocation request succeeded in the hope that subsequent requests for the same will be satisfied by the same Segment Tree. This may not always be true, but works well for many practical workloads.

All these optimizations greatly reduce the constants and asymptotic complexity of each operation, and improve the performance of the allocator.

6 Debugging Aids

The older `bitmap_allocator` was pretty easy to corrupt (as so will the new one) since the bitmaps used to locate free objects lies just before the user memory. There were no checks performed to verify the sanity and integrity of the bitmaps.

The new allocator should have some sort or magic words on the boundaries of not just the bitmap, but also the boundaries of the memory region that stores user memory. Additionally, a debug mode could enable a more computationally intensive parity checksum on the bitmap.

7 Profiling Aids

In `Profile Mode`, the `bitmap_allocator` should be able to provide some information about the locality of the memory access patterns. As a first attempt, it could (in each Segment Tree) compute the quotient:

$$\frac{\#\ of\ allocated\ objects}{Total\ \#\ of\ objects\ between\ the\ first\ and\ the\ last\ allocated\ object}$$

and determine the *tightness* of the active set. The higher this ratio (closer to 1), the better the performance of the application.