AppShift MemoryPool: Most extensive and efficient MemoryPool

Sapir Shemer

January 14, 2022

Abstract

The aim of this paper is to introduce new concepts for memory pools — heap scoping and MemoryPool architecture automation, while also discussing previous literature and summing up known concepts and dilemmas in a single, concise paper. Ideas and algorithms elaborated in this paper are also implemented in a new open-source MemoryPool that integrates the known and new technologies and approaches into one single smart pool.

Contents

1 Introduction

Recent profiling in Google has concluded that almost 7% of all CPU cycles in their data centers is spent on memory allocations [1]. In retrospect, previous studies have shown that dynamic memory management contributes to processing latency an average of 30% and up to 43% of the processing time [2][3][4][5][6]. Following said results studies and work have been done to describe memory managers and introduce memory pool techniques.

AppShift::MemoryPool's implementation focuses on userland (*i.e.*, user space) memory allocators, as they do not allocate new space by calling the kernel such as the well-know *malloc(3)*, instead they create reusable blocks of dynamic memory. As a result, allocators of this kind can save CPU cycles and a context switch by pre-allocating ready to use space from the heap and not making a system call [7].

My approach in the implementation of the memory pool is based on a data oriented paradigm. Memory pools, in essence, manage pre-allocated blocks in the heap, those blocks have a unique structure that supports the allocation and deallocation mechanism that the pool provides and saves necessary metadata in those structures. Each structure in the memory pool will be implemented by a *struct*, the functions responsible for working with those structures will be part of a *MemoryPool* class. Working with this approach I get a clean and intuitive separation between our data and the operations that I make on the data.

Using different structures to create a hybrid memory pool can be highly useful. Ben Kenwright, in his paper *Fast Efficient Fixed-Size Memory Pools*, starts with mentioning a hybrid solution, then moves to assume that using multiple fixed-size pools can produce a general solution to replace the current system memory manager [8]. This assumption can be dangerous in some cases. In fact, Johnstone's and Wilson's analysis in their paper *The Memory Fragmentation Problem: Solved?*, found out that on average 90% of a program's allocations are of only 6.12 sizes, while the rest fall into categories of about 178.9 sizes[9]. Allocating and handling about 190 fixed-size memory pools can have a toll of its own, thus a fitting memory pools architecture should be chosen.

As mentioned, the paper will describe different structures for more than one type of pool, I take both fixed-size pools and general purpose pools into consideration. Depending on the context, one type of structure might be preferred to another, and selecting the best pools for a given software shall be referred to as selecting the MemoryPool architecture. Therefore, I provide algorithms to systematically analyze allocations of a software in run-time, then evaluate the best architecture based on the outcome. I will integrate those algorithms into our memory pool to create a memory manager smart enough to self-determine the best architecture.

Integrating the memory pool manager and structures that I provide can be done by simply overriding the *new* and *delete* operators by the memory pool's allocate and free operators. As a result, every C++ allocation will be handled by the memory pool manager, allowing it to analyze the software to implement the best architecture and even speed up the standard library itself.

2 Memory Manager: Rigorous definition

Rigorously defining a memory manager using a mathematical language can help us construct a model for proof and a more precise description for our implementations. Memory managers are known to work on memory as the name implies, they allow us to manipulate memory for use so that we know where we can store our data.

S_1	S_2	S_3	$\cdots$	S_N

We first need to describe what is memory. In a strict sense, at any given moment in time, a memory contains a set of symbols that is currently saved in the memory (might even contain an empty symbol or 0), which is the state of the memory. Memory that is segmented into bits has a state in the set $\mathbb{B}^N$, such that $\mathbb{B} = \{0, 1\}$, and N is the size available in our memory.

Memory Definition Given that M is a memory, Σ is the set of the available symbols, Δ is set of addresses with ordering $\prec$ and size N, then the following properties apply:

1. $M(t) : \Delta \to \Sigma^N$ Where t is a given point in time. In other words we have $M : T \to \Sigma^N$, where T is the set of points in time.

2. A **memory segment** M_i^j is also a memory with symbols Σ and length j, where for any t:

$$M_i^j(t) = (\pi_i(M(t)), \ldots, \pi_{i+j-1}(M(t))) \in \Sigma^j$$

Concatenating Functions Given 2 maps $f : M \to B$ and $g : N \to C$, then their concatenation $f \odot g : (M \cup N) \to (B \cup C)$ implies the following:

1. For every $a \in M/N$: $(f \odot g)(a) = f(a)$
2. For every $a \in N$: $(f \odot g)(a) = g(a)$

Proposition 1 For any natural numbers i, j, k, and whole number c where $\min\{j, k\} \geq c \geq 0$ the following holds for any point in time $t \in T$:

$$M_i^{j+k}(t) = M_i^j(t) \odot M_{i+j-c}^{k+c}(t)$$

3 Structure: Segregated or Stack?

A memory pool can be structured in one of two ways — segregated and stack based:

- **Segregated Pools** are pre-allocated blocks which are divided into fixed-size chunks that are used for allocation, thus they are very fast and fragmentation is not much of a problem as the fragments are the first chunks to be allocated, and in constant time *O(1)*.
- **Stack pools** are pre-allocated blocks that do not care about any fixed-size, they allocate the same way as the stack does hence their name, they are very efficient when working with varying sizes.

Stack pools can have more complexities than segregated pools. In fact, for a stack based pool a variety of different algorithms for fragmentation handling and garbage can be chosen, or even completely discarded which can balance time and space efficiency in the relevant context.

In a strict sense, a memory pool is more abstract, as it is just a pool of pre-allocated space with an allocation/deallocation mechanism, therefore a single memory pool can even consist of more than one chain which can be of a different structure each. I shall refer to segregated and stack based blocks as *atomic pools*, a linked list of those blocks as *chain pools*, and pools with more than one chain or block as *complex pools.*

Both segregated and stack based pools create pre-allocated blocks that might be chained (*i.e.* as a linked list) for adding more space for further allocations. This does not necessarily mean that it is a good idea to make a chain pool from stack pool blocks and segregated pool blocks together, as their internal structure and logic differ significantly. As a consequence such chains can make a mess, thus it is better to create each chain of only same types of blocks.

Furthermore, making a segregated chain pool with blocks containing different fixed-size chunks can also add a toll when trying to find the block with the relevant size when requesting an allocation. Efficiently handling chains of segregated pools is done by making a chain of blocks for every fixed-size that is allocated, then managing them together manually when writing allocation or as a map from the chosen size to the fitting fixed-size pool for automation.

3.1 Segregated Pool

Chain pools of segregated blocks have bigger headers than atomic pools as they need to contain metadata for the next and previous blocks, even every chunk in a segregated chain pool has a header that points to the block header that contains the chunk, while a chunk in an atomic pool of a segregated block doesn't necessarily need a header. I take into consideration only a chain pool block header, as an atomic pool block can be evaluated trivially by removing unnecessary data.

```
struct SSegregatedBlock {
    // Block metadata
    size_t size;
    size_t chunkSize;
    size_t offset;

    // Movement to other blocks in chain
    SSegregatedBlock* next;
    SSegregatedBlock* prev;

    // Memory fragments handling
    SSegregatedDeleted* lastDeletedChunk;
};
```

3.1.1 Chunks

Segregated chain pools are divided into fixed-size chunks, each chunk has its intended size with an additional header. The header helps keep track of each chunk by indicating which block it resides in — this makes deallocation a simple constant time process. The attentive reader might notice that in case of an atomic pool this header is not necessary, Kenwright implemented a great example for such a pool [8].

```
struct SSegregatedChunk {
    // Pointer to the block containing the chunk
    SSegregatedBlock* container;
};
```

As you can notice in the segregated block header, I keep the *lastDeletedChunk* to keep track of the last chunk that was deleted. Deleted chunks have their own unique headers, each of which is pointing to the previous deleted chunk. In this manner, when freeing a chunk I just replace its header to point to the previously deleted chunk, then update our *lastDeletedChunk* to match the new deleted chunk. In fact, when requesting space from the segregated pool, I just need to first check with the *lastDeletedChunk* that is not *null*, if it is indeed not *null* then I can return it as a newly allocated chunk, and update the *lastDeletedChunk* to the previous one.

```
struct SSegregatedDeleted {
    // Pointer to the previously deleted chunk
    SSegregatedDeleted* previous;
};
```

3.1.2 Algorithms

You might have noticed that I introduce a variable named *offset* into the segmented block header, which allows us to keep track of the last chunk that was allocated. Efficient memory consumption in this manner is possible without adding any overhead. In a strict sense, when allocating I first prioritize releasing fragmented memory, and only if I don't have any fragments then I add to the top of the block and update the index accordingly. Finally, if there is no memory in the garbage, and the offset equals the size, then I create a new block to the end of the chain and return its first value.

Allocate The following algorithm is implemented in the *MemoryPool::allocateSegregated* function:

1. If *lastDeletedChunk* is not *nullptr*, then return it as an allocated chunk.
2. If *offset* == *size*, then create a new block in the chain and return its first chunk.
3. Return the next chunk that is next the *offset*, and increment it by 1.

Free The following algorithm is implemented in the *MemoryPool::freeSegregated* function:

1. Retrieve block from chunk header by subtracting the header's size from its pointer.
2. If at offset, then decrement offset and return.
3. Interpret chunk header as a deleted header, and point to *lastDeletedChunk*.
4. Set *lastDeletedChunk* at the new deleted header.

3.2 Stack Pool

```
struct SStackBlock {
    // Block metadata
    size_t size;
    size_t offset;

    // Movement to other blocks in chain
    SStackBlock* next;
    SStackBlock* prev;

    // Free vs Occuppied space metadata
    size_t numberOfAllocated;
    size_t numberOfDeleted;
};
```

3.2.1 Chunks

```
struct SStackChunk {
    size_t size;                // Allocated space
    SStackBlock* container;     // Block nesting the unit
};
```

```
struct SStackDeleted {
    size_t size;                     // Allocated space
    SStackDeletedUnit* previous;     // Block nesting the unit
};
```

3.2.2 Algorithms

4 Efficiency: Space or Time?

4.1 Performance mode

4.2 Scoping The Heap

```
struct SMemoryScope {
    // Pointer to block of scope
    void* container;
    size_t offset;
    // The parent scope for nesting scopes
    SMemoryScope* parent;
};
```

References

[1] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[2] Paul Wilson, Mark Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. *In: International Workshop on Memory Management*, 986, 10 1999.

[3] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive c programs. *SIGPLAN Not.*, 27(12):71–80, dec 1992.

[4] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between c and c++ programs. *Journal of Programming Languages*, 2, 02 1994.

[5] Wentong Li, Saraju Mohanty, and Krishna Kavi. A page-based hybrid (software-hardware) dynamic memory allocator. *Computer Architecture Letters*, 5:13 – 13, 03 2006.

[6] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. *SIGPLAN Not.*, 37(11):1–12, nov 2002.

[7] Martin Kočička. Performance analysis of the lsu3shell program. Master's thesis, Faculty Of Information Technology CTU In Prague, 2019.

[8] Ben Kenwright. Fast efficient fixed-size memory pool: No loops and no overhead. *COMPUTATION TOOLS*, 2012.

[9] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? *SIGPLAN Not.*, 34(3):26–36, oct 1998.