CHALMERS

JEFF and STEVE: Allocator and Analysis Tool

The Tales of The Compacting Allocator JEFF and His Friend STEVE, The Malloc Benchmarking and Analysis Tool

Master of Science Thesis in Computer Science

MIKAEL JANSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, April 2015

JEFF and STEVE: Allocator and Analysis Tool
The Tales of The Compacting Allocator JEFF and His Friend STEVE,
The Malloc Benchmarking and Analysis Tool

MIKAEL JANSSON

Examiner: KOEN LINDSTRÖM CLAESSEN

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden April 2015

Abstract

Jeff is a compacting allocator and Steve is an allocator benchmarking tool. Steve can be used to benchmark any application with any allocator. Steve records memory traces of an application during execution and uses those traces to benchmark any number of allocators for which there are drivers. Therefore, the performance in actual use cases is measured. Also, as a consequence, the user does not need access to source code to the application or allocators to be tested. Users can easily write their own allocator drivers to extend Steve with. Compared to the tested allocators in this report, Jeff performs similar or better in terms of speed, but similar or worse in memory.

Contents

Acknowledgements

Thanks to Anders Höckersten and Peter Wallman at Opera for brainstorming, and to Kristian Wiklund for comments and suggestions. Thanks to my supervisor Koen Lindström Claessen at Chalmers University, my friends and family for being understanding during all the time I've spent on this thesis.

Mikael Jansson Chalmers, April 2015

Ethical Considerations

No ethical problems were identified during the course of the work. The nature of the work does not include any experiments nor studies on humans or animals.

Chapter 1

Introduction

Computer systems can be generalized to be composed of two things: data, and code operating on said data. In order to perform useful calculations, real-world applications accept user data which often varies in size. To accommodate the differences, memory is requested dynamically (at runtime) using a memory allocator. The basic interface to the allocator consists of the two functions *malloc(size)* and *free(pointer)*: the application gives malloc a size and retrieves a pointer to a chunk of memory guaranteed to be at least *size* bytes. The operation *free(pointer)*, on the other hand, gives the memory back to the system.

The allocator in turn calls out to the operating system-provided memory mapping function, providing the allocator with one or more *pages* of memory. The page is the smallest unit of memory available to the operating system from the processor and its memory mapping unit (MMU), and in the architectures widely used today (x86, x86_64, ARMv7, PPC) the default size is 4 KB, and can on some architectures be increased.

On operating systems where application memory is protected from other applications' memory, meaning no application can overwrite any other application's memory, the page addresses are virtual and therefore a kernel-space[1] look-up table mapping virtual (page) address to physical memory is required, such that each process has its own look-up table. Since the processor needs to keep track of each page and to which part of memory it is mapped, the resulting look-up table will be very large if a small page size is used. These can then be written to disk (*swapped out*) when system memory becomes full and the pages are not in use by the application owning the page, based on a recently-used algorithm. The algorithm varies depending on operating system and use case. In the worst case, the last page in a series of requested pages is largely unused, causing 4096-1 bytes to go to waste. Keeping the page size small lowers this waste, which is also known as *internal fragmentation*. Increasing the page size can be tempting to speed up the lookup table, but this comes at the cost of fragmentation. In some architectures, the operating system can increase the page size (so-called "huge pages" in Linux terminology, other operating systems use different names).

A very simple user-space[2] allocator would do little more than *malloc()* returning the pages and have *free()* do nothing at all. Clearly this causes large

[1]The operating mode of the operating system where the kernel and hardware drivers run.

amounts of memory to be wasted, since no memory would actually be released to the system. Eventually, the system would run out of memory.

Therefore, an allocator needs to be more clever about managing memory. At the very minimum it needs to associate metadata with each allocated block in order to later free the blocks. The metadata is there in order for *free()* to know where the memory chunk was allocated from. Moreover, the allocators I've tested keep one or more pools of memory split up in different size ranges, such as "small blocks", "medium-sized blocks" and "large blocks". By analyzing the runtime requirements of various applications, the most commonly use cases, such as a specific block size dominating all other requests, can be optimized for speed, space or both. The pool(s) are used because of the fact that applications often allocate data of particular sizes.

If the allocator can group together all e.g. 8-byte chunks into one or more pages, it will be easier to return the page to the operating system when all blocks are freed. Lookup can also be more efficient, since the allocator can use offsets to find a suitable free block, instead of iterating through a list of free blocks.

1.1 Thesis Statement and Contributions

The purpose of this thesis is to design and implement an allocator with the following interface that can move around allocated blocks of memory:

```
handle_t malloc(size_t n);
void *lock(handle_t h);
void unlock(handle_t h);
void free(handle_t h);
```

The purpose of the lock/unlock operations is to introduce indirection for memory access that gives the allocator the ability to move data blocks around when not in use (*unlocked* state), specifically by compacting the heap cope with fragmentation problems.

In Chapter 2 I present an overview of the allocator I compare my work with. Method and Design are in Chapters 3 and 4.

I have developed a method of simulating runtime behaviour of application using heuristics and show that it is possible to test performance of locking/unlocking allocators without access to source code. This is done in Chapter 5.

I show that randomized testing in large volume is a useful technique for finding problems in complex data structures, such as an allocator. This is done in Chapter 6.

I have collected a variety of applications that can be modified to use the different allocation interface for benchmarking purposes. This is done Chapter 7. The results from benchmarking the allocators, can be found in Chapter 8 which is finally discussed in Chapter 9.

[2]The operating mode of the operating system where normal applications run.

1.2 Definitions

- **Opaque type**: A way of hiding the contents of an object (data structure) from application code, by only providing a pointer to the object without giving its definition. Commonly used where the object is only meant to be modified from the library.
- **Valgrind**: A debugging tool used to detect memory leaks and memory overwriting in applications, by emulating the target CPU.
- **mmap()**: A system call for applications to ask the operating system for one or more memory pages (often 4KB) and map it into the application's virtual address space.
- **sbrk()**: Similar to *mmap()*, but works by extending the application's data segment size instead of asking for virtual memory, and is limited by the maximum data size of the application.
- **Internal fragmentation**: The amount of memory wasted inside a block.
- **External fragmentation**: The amount of memory wasted by allocator metadata.
- **Memtrace**: File created by Valgrind's *memcheck* tool (see Chapter 7) that contains triplets of *(op, address, size)*. See the appendix for the full definition.
- **Op**: Any memory operation: new, free, load, store, modify, lock, unlock. Generally, load, store and modify is generalized to access. These are sometimes abbreviated to N for new, F for free, A for access, L for load, U for unlock.
- **Opsfile**: File created by `translate-memtrace-to-ops.py` (part of Steve, see Chapter 7), contains one operation per line. See the appendix for the full definition.
- **Block**: A chunk of allocated memory.
- **Lifetime**: The number of total operations, thus indirectly the time, between a New and a Free op for a specific block.
- **Header**: An internal data structure containing the metadata about a block. It is also used as an opaque handle for use by the client code.

1.3 Challenges

There are many trade-offs when writing an allocator, which I'll describe in the following section.

Allocators are often optimized for a specific use-case or task, while still performing well in the average case. In fact, some allocators are designed with the explicit goal of being best on average.

A very simple allocator would simply request one or more pages from the operating system and return in to the user. It would be very fast, but not very efficient since a large part of the page would be unused for any allocation requests smaller than the page size.

By splitting up allocations in parts exactly the size of the requested block (plus metadata) and storing information about freed blocks in a list, there would be little wasting of memory. On the other hand, because of the efficiency requirement, pages would only be requested when there were no blocks of the correct size and therefore the entire free list must be searched for a suiting block before giving up and requesting a page.

Multi-threaded applications that allocate memory need to work without the allocator crashing or corrupting data. As in all concurrency situations, care needs to be taken to do proper locking of sensitive data structures, while not being so coarse such that performance suffers. I do not address the issue of locking.

Another challenge is to make the allocator work efficiently for various memory sizes. I focus on small-memory systems, where space-efficiency is important, and I've made the trade-off (where applicable) that slower is better if it saves memory. It is currently in use at TLab West Systems AB on an embedded computer with a total of 512 KB RAM.

1.4 Efficiency

Around 2007 at Opera, a company that produces a web browser for desktop computers, embedded computer systems and phones, memory fragmentation became a problem after repeatedly loading and unloading web pages. Large web pages load many small resources, specifically images, that create holes in memory when freed. After a few page loads, it is no longer possible to load any more pages because there are no continuous blocks of memory large enough to fit a web page in. It happened frequently on small-memory devices, such as early smart phones and feature phones with 4-8 MB RAM.

Because of said fragmentation, large enough blocks can eventually not be allocated, even though the total amount of free memory is greater than the requested block size. This goes against the findings by Johnstone & Wilson (1998), where in the average case, the level of fragmentation is good enough. However, for Opera, "good enough" was insufficient. By making a custom allocator with the signature outlined in the hypothesis, they hoped to solve the fragmentation problem in the specific situations that occur in a web browser. Another use case for the allocator was for an in-house custom programming language, where the allocator's purpose was to be used as a garbage collector. This did not happen, however, because of delays in finishing the thesis.

1.5 Related Work

Closely related to a compacting allocator is the garbage collector, which is popular in managed languages that do not run directly on hardware. In particular, the Java Virtual Machine (JVM) includes different garbage collector (GC) flavors depending on the task at hand. As of version 5, there are four variants (Sun Microsystems, 2006) with different characteristics that can be picked depending on the type of application written. Each GC flavor can be configured.

Configuration settings, including setting GC flavor, can be done at runtime via command line parameters to the JVM.

All JVM GCs use *generations*, in which objects are allocated and later moved if they survive a garbage collection. This is mainly done as an optimization to execution time since different collection strategies can be used for "young" objects and "old" objects (i.e. the ones that have survived a set number of collections). A generation is implemented as separate memory areas, and therefore, areas that are not full waste memory. Also, application code is unaware of when collection occurs, generations is also a means of reducing the time the application is paused, if the collection cannot happen simultaneously with application execution. Pausing in general is a problem GCs try to solve, see Jones & Lins (1997).

In my thesis, I give control over pausing to the application that can decide at its own discretion when the most appropriate time is for heap compacting. In the optimal case, where a simple *bump-the-pointer* technique can be used, i.e. increase the heap pointer for the next chunk of memory, allocation will be very quick, at the expense of compacting having to occur frequently. This is a deliberate trade-off, based on the assumption that there will be idle time in the application where compacting is more appropriate. Generations would be of no benefit in this scenario. In the worst case, however, blocks on the heap are locked at compacting time. These blocks cannot be moved and therefore a free list needs to be maintained, causing allocation to be slower. Being able to move these blocks to a location where they cause less harm is left as future work.

Chapter 2

Allocator Types

In this chapter I'll describe the most common styles of allocator implementation strategies.

2.1 Buddy Allocator

The most common allocator type is the buddy allocator[3], and many allocators are built on its principles, or at least incorporate them in some way: start with a single block and see if the requested chunk fits in half of the block. If it does, split the block into two and repeat, until no smaller block size would fit the request. The allocator described in this paper does away with merging blocks directly on free, by moving free blocks together and coalescing them in the compacting step.

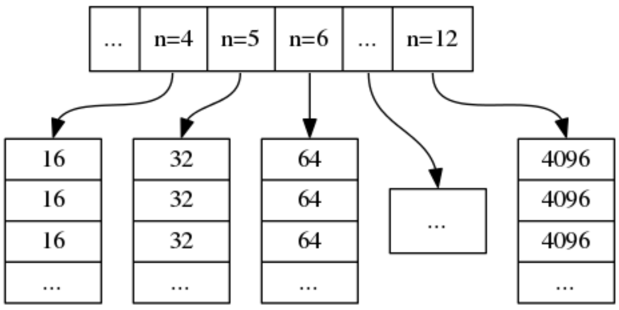


Figure 2.1: The free list in a buddy allocator

Over time, there will be more and more items of size 2^n, stored on a free list for that block size, as shown in Figure 2.1 Each pair of split-up blocks is said to be two buddies. When two buddy blocks are free, they can be joined. A block of the next larger size $(n + 1)$ can be created from these two blocks. The merge repeated until the largest block, i.e. 2^k. In the worst case, this causes $2^n - 1$

[3]http://en.wikipedia.org/wiki/Buddy_memory_allocation

bytes of overhead per block, also known as *internal fragmentation*. Still, this commonly used algorithm has shown to be good enough in many cases and is often incorporated as one strategy of an allocator.

All blocks must have metadata associated with them. The minimum amount of information is the length of the block, in order for *free()* to know where the block ends. For convenience, this is often stored in memory just before the block itself. The metadata could also be stored elsewhere, e.g. a lookup table *S(addr)* that gives the size of the block starting with *addr*.

The metadata associated with the block is normally not accessible by user code (unless queried using allocator-specific debugging code, when available) and adds to external fragmentation. It is the fragmentation between the user blocks (hence *external*), i.e. any overhead caused by information required by the allocator, but not the user code.

Conceptually, the buddy allocator is a very simple allocator to use and implement, but not the most efficient because of internal fragmentation.

2.2 Pool Allocator

Certain applications use a large amount of objects of the same size that are allocated and freed continuously. This information can be used to create specialized pool allocators for different object sizes, where each pool can be easily stored in an array of $N * sizeof(object)$ bytes, allowing for fast lookup by malloc and free. For example, in an action computer game where both the player and the enemies shoot projectiles with weapons, said projectiles must be kept track of by means of an object in memory. In a game where there could potentially exist a very large amount of projectiles in action at the same time, an equal amount of allocation and freeing is done, often randomly. In order to optimize usage of system memory, all allocation of projectile objects would be contained to the same *pool* of memory. A very simple such allocator would be a simple list of objects and the allocator could just return the index to the next unused block, leading to malloc and free that both have $O(1)$ in time complexity with very little overhead and fragmentation. The usual strategies for growing the list apply, such as doubling the size of the list when all items in the list are used.

2.3 Arena Allocator

Code with data structures that are related to each other can be allocated from the same arena, or memory region, for example a document in a word processor. Instead of allocating memory from the system, the allocator requests data from a pre-allocated larger chunk of data (allocated earlier from the system). The main point of this type of allocator is quick destruction of all memory related to the working data (i.e. the document), without having to traverse each individual structure associated with the document to avoid memory leaks from not properly freeing all memory. In applications where a document/task has a well-defined set of objects associated with it, with relatively short lifetime of the document, but many documents created/destroyed over the total application lifetime, there is a large speed benefit to be had from making the free operation faster.

2.4 Garbage Collector

A garbage collector is an allocator that automatically provides memory for data as needed. There is no need to explicitly ask for memory from the allocator, nor to free it when done. Instead, the garbage collector periodically checks for which objects are still in use by the application (*alive*). An object is is anything that uses heap memory: a number, a string, a collection, a class instance, and so on. There are several techniques for finding alive objects and categorizing objects depending on their lifetime in order to more efficiently find alive objects at next pass. This is described in detail by Jones & Lins (1987).

At any point in time, a garbage collector can move the object around if necessary, therefore any object access is done indirectly via a translation. User code does not keep a pointer to the block of memory that the object is located in, but instead this is done with support in the programming language runtime.

A garbage collector, because of the indirect access to memory, can also move objects around in a way that increases performance in different ways. One way is to move objects closer together that are often accessed together, another way is to move all objects closer to each other to get rid of fragmentation, which would decrease the maximum allocatable object size. A normal allocator cannot do memory defragmentation, or any other memory layout optimization, because memory is accessed directly, which would invalidate the pointer used by the application code.

Chapter 3

Method

The method I used for designing and implementing the allocator (Jeff) is an iterative design process based on experimental designs and verification thereof, along with theoretical calculations using pen and paper. In particular, the compact operation went through several iterations in my sketch book before I arrived at the final version. For other parts, I used the profiling tools found in *GCC* to measure bottlenecks and gradually improve the code until there were no obvious bottlenecks left. A bottleneck in this case is a piece of code that gets called many times and is slow. I've aimed to write the code to execute reasonably quick, given the algorithm in use.

Thanks to the rigorous testing framework in place, I could readily modify code without fearing malfunction since the tests would pick up any errors. A limitation of testing is that it can never prove correctness, only absence of the bugs the testing framework was designed to find. The testing framework is described in more detail in Chapter 6.

Steve is the name of the benchmark tool that I designed to test algorithms for Jeff and to compare Jeff to other allocators. In Steve, I've developed heuristics for calculating locking/unlocking based on runtime data of unmodified applications. The tool for doing so grew from a small script into a larger collection of tools related to data collection, analysis and benchmarking.

Data for use by Jeff, and other allocators, is collected by various parts of the benchmark tool. The types of data used are:

- raw memtrace, from running unmodified applications in my modified Valgrind tool
- ops file, by mapping memory access data to objects
- locking ops file, as above, with lock/unlock operations in place
- benchmark output, by running allocators against ops files

The benchmark output is used to both produce graphs of allocator performance and can together produce internal rankings between the allocators.

This is described in greater detail in Chapters 5, 7 and Section 8.2.

3.1 Assumptions

- A1: An allocator with little extra increase in memory usage compared to the requested memory by the client application is efficient in space.
- A2: An allocator that has a small and preferably constant execution time is efficient in time.

3.2 Hypothesis

- H1: An allocator that performs heap compaction can be efficient in both time and space, compared to other commonly used allocators. By making the malloc and free operations fast and the compact operation relatively slow and calling it when the system is idle it is possible to achieve this. On memory-restrained and slow computer systems, such as embedded systems with as little as 512 KB RAM and a 50 MHz CPU, it is important to be efficient in both time and space.

What are the space and time requirements of Jeff compared to other popular allocators? Is Jeff a viable alternative to other popular allocators in real-world situations?

I aim to answer these questions in the report in Chapter 8 where I display both graphs and runtime numbers, and in Chapter 9 where I assemble the results from the allocators I've measured.

3.3 Development Environment

The main development system is a Linux-based system (32-bit Ubuntu 13.04), but could likely be adapted to other UNIX-like systems, such as OS X. Porting it to 64-bit systems requires changing the manual type casting from and to integer types that assumes a pointer will fit in a 32-bit integer.

The allocator is written in C, and the benchmark tool (Steve) consists of mostly Python code with Cython[4] (a Python framework for interfacing in C and compiling Python into C code) for tight inner loops such as the memtrace to ops calculation, plus some Bash scripts for glueing it all together. The data is plotted in graphs and there is also a tool that creates an animation of memory allocations as they happen in memory.

Parallel with allocator development, I wrote tests using Google's C++ testing framework, googletest[5], to make sure no regressions were introduced during development. More on that in Section 3.4.

3.4 Testing

All applications should be bug-free, but for an allocator it is extra important that there are no bugs since an allocator that does not work properly could cause data corruption. In the best case, this causes the application using the allocator

[4]`http://cython.org`

[5]`http://code.google.com/p/googletest/`

to malfunction by crashing on execution. In the worst case, an application doing data processing by reading data into buffers allocated on the heap, doing one or more computations and then writing the data back to disk, would completely destroy the data without the user knowing an error had occurred.

Luckily, an allocator has a small interface for which tests can be easily written. In particular, randomized unit testing is easy, which gives good coverage.

I decided to use googletest since it was easy to set up and use, and the results are easy to read. It's similar in style to the original Smalltalk testing framework SUnit[6] (later popularized by Java's JUnit[7]). During the development of the allocator I wrote tests and code in parallel, similar to test-driven development in order to verify that each change did not introduce a regression. Of the approximately 2500 lines of code in the allocator, about half are tests. In addition to randomized unit testing there are consistency checks and asserts that can be turned on at compile-time.

In the unit tests, the basic style of testing is to initialize the allocator with a randomly selected heap size and then run several tens of thousands of allocations/frees and make sure no other data was touched. This is done by filling the allocated data with a constant byte value based on the address of the returned handle. Many bugs were found this way, many of them not happening before thousands of allocations were made.

[6]`http://en.wikipedia.org/wiki/SUnit`

[7]`http://en.wikipedia.org/wiki/JUnit`

Chapter 4

Design

The implementation of the allocator (Jeff) is described in detail in Chapter 6, and the implementation of the benchmark tool (Steve) is described in detail in Chapter 7.

4.1 Design Background

To get started with my allocator, I started implementing a buddy allocator since the basics of the buddy allocator is used in the allocators I've included in my comparison, (Evans 2006). Since a buddy allocator's two main modes of operation are splitting and joining blocks, care needs to be taken that these two operations are as quick as possible.

During development, I quickly realized the fact that picking the wrong data structure for storing the block list made splitting and joining operations slow and error-prone.

I therefore discarded the buddy allocator prototype and started on the actual allocator that was to be the end result, with lessons learned from the prototype incorporated into the design phase.

4.2 Allocator Design

I assume that there will be idle time when the application is not doing any other processing. The functions malloc() and free() should perform their work as quickly as possible. If possible, any processing that doesn't have to be done immediately, should be done in the application's idle time. An example of such a task is compacting. Moreover, since I have the freedom to move objects around in memory transparently for client code, any logic in free/malloc that handles memory layout, e.g. for optimization purposes (either space or time efficiency) should be handled in the idle time to as a high extent as possible.

From this, we arrive at my original idea of a quick malloc, a quick free and a slow compact, the latter performing any batch processing postponed from free and malloc. I envisioned this as a malloc that would basically grow the top pointer of the malloc-associated heap: store information about the requested block size, increase the top pointer and return the block. The free operation would mark the block as not in use anymore. Eventually, the top pointer would reach the end of the heap, at which point the compact operation would go through the heap and reclaim previously freed memory and reset the top pointer to end of allocated memory leaving the freed memory as a large block of memory ranging to the end.

However, that idea turned out incorrect because blocks can be locked at the time of compact. Recall that locking a block gives the client code the actual pointer to memory, and unlocking the blocks invalidates the pointer. Therefore, the worst-case scenario is that a block at the very top of the heap is locked when the compact operation is invoked. Even though all unlocked free blocks are coerced into a single free block, a locked block at or close to the top would make subsequent malloc calls to fails. Therefore, a free list needs to be maintained even though it might be the case for real-world applications that the worst case seldom occurs. I have not studied the frequency of this happening in this report.

4.2.1 Free In More Detail

When an allocation request comes in, the size of the request is checked against the top pointer and the end of the heap. A request that fits is associated with a new handle and returned. If there is no space left at the top, the free list is searched for a block that fits.

Freeing a block marks it as unused and adds it to the free list, for malloc to find later as needed. The free list is an index array of $2^{3..k}$-sized blocks with a linked list at each slot. All free blocks are guaranteed to be at least 2^n, but smaller than 2^{n+1}, bytes in size. The blocks can then be reused in malloc directly, or merged together all at once in the compacting operation. This is unlike the buddy allocator where blocks are merged directly on free. (See future work in Section 9.4.2 for a brief discussion on merging directly on free.)

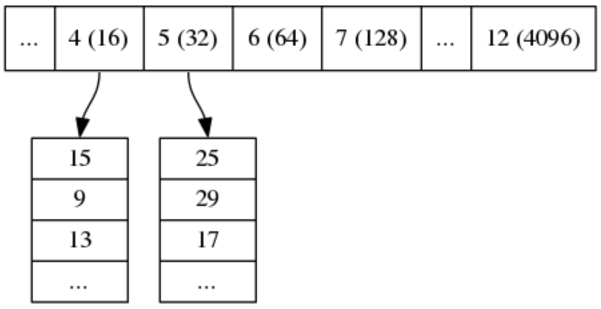


Figure 4.1: Example slots in free list.

An example free blockslots list is given in Figure 4.1.

4.2.2 Compacting

Compacting uses a greedy Lisp-2-style compacting algorithm (Jones & Lins (1997)), see Section 4.3.4 for a step-by-step version and Section 6.2.4 for an explanation with figures. In short, blocks are moved closer to bottom of the heap if possible, otherwise the first block (or blocks) to fit in the unused space is moved there. The first case happens if there are no locked blocks between the unused space and next used (but not locked) block, simply moving the memory blocks and updating pointers is enough. A quick operation that leaves no remaining holes. If however there are any locked blocks between the unused space and the next used block, obviously only blocks with a total length of less than or equal the size of the unused space can be moved there. The algorithm is greedy and takes the first block that fits. More than one adjacent block that fits within the unused space will be moved together. In the case that there are no blocks that fit the unused space and there is a locked block directly after, scanning is restarted beginning with the block directly following the last free block found. The process is continued until there are no unused blocks left or top of memory is reached.

4.3 Allocator Algorithm

4.3.1 Initialization

1. We're passed a heap of a given size from the client
2. Set boundaries of the header list growing down from top of heap
3. Initialize the free block slot list

4.3.2 Allocation Request

1. Request a new header to associate with the block
    1. If built with unused header list, grab the first one in list and relink root
    2. Else, scan the header list for unused header. If not available, move bottom down one header.
    3. If bottom clashes with space occupied by a block, fail.
2. If there is available space for the allocation request, use it and associate with the block.
3. Else, find a free block within the free block slot list:
    1. Search in the slot associated with the log_2-size of the request for a free block.
    2. If not found: repeat the previous step in higher slots until top is reached. If there are still no blocks found, fail.
4. Split the block as needed, insert the rest into the free block slots and return the rest.

4.3.3 Free Block

1. Mark the header as free

2. Overwrite the block with a free memory block structure pointing to the header location, with the struct's memory member pointing to `NULL`.
3. Insert the block into the appropriate location in the free block slots list.

4.3.4 Compact Heap

1. Sort the header list items' next pointers in memory order.
2. Starting from bottom of the heap: while there are unoccupied spaces in the rest of the heap or compacting has reached its time limit, do the following.
3. Scan for the first unlocked[8] memory block.
4. If there are no locked blocks between the unoccupied space and the first unlocked memory block, move the memory by the offset between locked and unused memory.
5. If there are any locked blocks in-between, move only as much memory as will fit into the unlocked space. Create a free block of the rest of the memory inside the unoccupied space.
6. Restart from point 2.
7. Merge all adjacent free blocks and mark the headers not in used as unused.
8. Rebuild the free block slots by scanning the free header blocks and inserting them at the appropriate locations in the list.

4.4 Benchmark Tool Design

Manually modifying applications to adhere to Jeff's allocation interface is error-prone and time consuming, and moreover it is not certain that the chosen application is a good candidate for demonstrating performance since it might not stress the allocator. The number of requests could be small and the total memory usage could be low.

Measuring Jeff requires a rewrite of the application needing to be tested, to use the new malloc interface. The simple solution to do so is to emulate a regular malloc, i.e. directly lock after malloc. But that would make the compact operation no-op since no blocks can be moved. It is also not obvious which applications would make good candidates. Automating the modifications, if possible, would save much time. Finally, source code to the applications would be required for manual adaptions, which is not always available. I have therefore not done any manual adaptions of an application.

The specifics of how data is collected can be found in Chapters 5 and 7.

[8]Only unlocked memory blocks can be moved. Clients have references to locked blocks which therefore cannot be changed.

Chapter 5

Simulating Application Runtime

As described in the introduction, it is not a practical solution to rewrite applications to use the new API. Therefore, I automated the task of finding approximate locking behaviour of applications. One way of doing that is to simulate an application, which requires at the minimum the application's memory access patterns, i.e. when the application reads from and writes to memory.

5.1 Gathering Memory Access Data

Simply getting malloc/free calls is trivially done by writing a malloc wrapper and make use of Linux' *LD_PRELOAD* technique for preloading a shared library, to make the applications use a custom allocator that can do logging, instead of the system allocator. This requires the application to use the system *malloc* and *free* to work, since calls to a custom allocator within the application cannot be captured. By pointing a special environment variable (`LD_PRELOAD`) to the location of a shared library prior to executing the application, any symbols missing from the main application (which in the normal case is, among others, *malloc* and *free*) are searched for in that library, and only afterwards are the system libraries are searched. This is called *dynamic* linking, where symbols in the application are linked together at runtime, as opposed to *static* linking, where all symbols must exist in the application binary.

Unfortunately is is not enough to simply get the calls to malloc and free. To get statistics on memory *access* patterns one needs to essentially simulate the computer system the application runs in. Options considered were TEMU[9] from the BitBlaze[10] project, but because it would not build on my Linux system, I evaluated Valgrind[11] and concluded that the support for instrumentation in Valgrind met my requirements.

Valgrind was originally a tool for detecting memory leaks in applications on the x86 platform via emulation and has since evolved to support more hardware platforms and providing tools for doing other instrumentation tasks. Provided with Valgrind an example tool, *Lackey*, that does parts of what I was looking for but lacks other parts. Instead I ended up patching the *memcheck* tool[12] to capture load/store/access operations and log them to file, if they were in the boundaries of lowest address allocated to highest address allocated. This will still give false positives when there are holes (lowest is only decreased and highest is only increased, i.e. only keeping a range of *lowest..highest* of used memory) but reduces complexity of tracking memory. It does not affect the end result, except for taking longer time to filter out false positives. Memory access is then analyzed offline. Note that it will only work for applications that use the system malloc/free. Any applications using custom allocators must be modified to use the system allocator, which generally means changing a setting in the source code and recompiling.

This is how the modified Valgrind that produces memtraces files, memtrace translation and locking calculation fits together:

```
#!/bin/bash

theapp=$1

if [[ ! -d "$theapp" ]]; then
    mkdir $theapp
fi
shift
echo "$*" >> ${theapp}/${theapp}-commandline
../../valgrind/vg-in-place --tool=memcheck $* 2>&1 > \
    /dev/null | grep '^>>>' > ${theapp}/${theapp}

python -u ../steve/memtrace-to-ops/translate-memtrace-to-ops.py \
    ${theapp}/${theapp}

python -u \
    ../steve/memtrace-to-ops/translate-ops-to-locking-lifetime.py \
    ${theapp}/${theapp}
```

See also Figure 7.1.

Beware that running larger applications through the memory access-logging Valgrind takes very long time, about 30 minutes on an Intel Core i3-based system to load `http://www.google.com` in the web browser Opera[13].

5.2 Translating Memory Access Data to Ops

The basis of all further data analysis is a *memtrace*, a file with the output produced by the patched memcheck tool in the following format:

```
>>> op address size
```

[9]http://bitblaze.cs.berkeley.edu/temu.html

[10]http://bitblaze.cs.berkeley.edu/

[11]http://valgrind.org

[12]https://github.com/mikaelj/rmalloc/commit/a64ab55d9277492a936d7d7acfb0a3416c098e81 (2014-02-09: "valgrind-3.9.0: memcheck patches")

[13]http://www.opera.com

where op is one of N, F, L, S, M for New, Free, Load, Store and Modify, respectively and size is how many bytes are affected by the operation (always 0 for F). The operation New has an address and size associated, and it's therefore possible to map memory access (L, S, M) to a specific pointer. This is done by creating a unique identifier and mapping all keys from *address* to *address+size* to that identifier. On free, conversely, all mappings in that address range are removed. At each access a list of tuples <id, access type, address, size> is recorded.

The output file (*opsfile*) has the following format:

```
<handle> <op> <address> <size>
```

This is done by the tools `memtrace-run.sh` and `translate-memtrace-to-ops.py`. It took some effort to figure out the best way to perform the translation, however. I'll discuss the effort below.

5.2.1 Linear Scan

My initial attempt was to scan through the entire list each time for each operation. The problem is that Python is very slow and uses too much memory, which my laptop with 4 GB of RAM and an intel Core i3 CPU can't handle - this only works for small-ish outputs. This because the list of handles is checked for each memory access, i.e. a $\sim$ 2000 entries list for each memory access ($\sim$ 500 MB), quickly becomes unusable. I tried various approaches, such as moving the code to Cython (formerly known as Pyrex), which translates Python code into C and builds it as a Python extension module (a regular shared library), but only doing that did not markedly speed things up.

5.2.2 Save CPU at the Expense of Memory

I eventually tried a mapping on the start and end addresses, where each access address would be decremented towards start and incremented towards end. Each address is checked against against a mapping from address to handle. If the value (i.e. the memory handle) of the mapping is the same, I know that memory access belongs to a specific handle. That is even slower than iterating through 2000 elements, because the hash has to be checked on average one lookup per allocated byte in the memory area, even though the time complexity is similar: *O(n*m + c)* - the constant makes it slower, assuming hash lookup is *O(1)* i.e. *c*.

Finally, I came up with a brute-force solution: hash all addresses within the requested memory area, from start to end, mapping each address to the corresponding memory handle. The complexity is *O(m)*, but blows up with a MemoryError at about 2 GB data read (out of 12 GB in total) My server with 8 GB RAM has swap enabled, but by default Ubuntu 10.04 LTS doesn't overcommit memory. Setting `/proc/sys/vm/overcommit_memory` to 1 effectively enables swap for application memory allocation. Using a 32-bit system to allocate data larger than 4GB doesn't work. I installed a 64-bit Ubuntu LiveCD on a USB stick and did post-processing from that end. Now I could successfully translate a memory trace run to an ops file, given a computer with a large amount of RAM.

5.2.3 More on Lifetime

The lifetime calculation could be more elaborate. For now the calculation is fairly naive in that it only checks for long-lived object lifetime ranges, but it could also be setup to scan for "sub-lifetimes", i.e. module-global. My guess is that it would look like the histogram data in Figure 5.1 below, but located in the middle. Calculating that would mean that start and end points for calculating lifetime would be sliding, such that end is fixed and start moves towards end, or the other way around, where start is fixed and end moves towards start. Storing each value takes up lots of memory and analyzing the end-result by hand takes a very long time since one'd have to look at each histogram. I've implemented a simpler version of this, described below in Section 5.4.

5.2.4 Performance Optimization of Lifetime Calculation

Recall from definitions, Section 1.2, that lifetime is defined as number of ops on own handle divided by ops for all other handles, for the given handle's lifetime. For example, let's say handle A is created at time 0, handle B is created at time 10 and handle C is created at time 20. They all live until time 100 and each have 100 ops, evenly divided throughout their lifetimes. The lifetimes of the handles are:

- A: 100 / (100 + 100) = 50%
- B: 100 / (90 + 100) = 53%
- C: 100 / (80 + 89) = 59%

Each handle is mapped to a tuple <own, others>, and for each operation either own or others is incremented, until the handle is freed, at which point it's moved to the set of inactive handles. This means going through all handles for each operation, which for smaller datasets is OK. Even removing duplicates (two successive ops on the same handle) is quadratic *O(m*n)* (m = ops, n = live handles) takes too long time.

Instead, keep a counter of ops so far (ops_counter) and for each handle, store the triple (handle id, # own ops, ops_counter at handle new) and increase the number of own ops correspondingly. When the handle is freed, calculate the "other ops" value as *others_ops = current ops_counter - own - saved ops_counter* An example example with each line defined as ops counter | set of alive | set of dead, action:

```
20 | {(a 5 0) (b 2 5) (c 10 7) (d 3 17)} | {}, (death b) =>
20 | {(a 5 0) (c 10 7) (d 3 17)} | {(b 2 20-5-2=13)}, (death a) =>
20 | {(c 10 7) (d 3 17)} | {(b 2 13) {a 5 20-5-0=15}, (death d) =>
20 | {(c 10 7) (d 3 17)} | {(b 2 13) (a 5 15) (d 3 20-17-3=0)},
     (new e) =>
25 | {(c 10 7) (d 3 17) (e 5 20)} | {(b 2 13) (a 5 15) (d 3 0)},
     (new f) =>
28 | {(c 10 7) (d 3 17) (e 5 20) (f 3 25)} |
     {(b 2 13) (a 5 15) (d 3 0)}, (death e) =>
28 | {(c 10 7) (d 3 17) (e 5 20) (f 3 25)} |
     {(b 2 13) (a 5 15) (d 3 0) (e 5 28-20-5=3}
```

At the end, any remaining live handles (due to missing frees) are moved to the dead set.

This algorithm is *O(m) + O(n)*, down from *(Om*n)*.

5.3 Lifetime Visualization

A block with a lifetime close to the total number of operations is considered to have a long lifetime and therefore created in the start of the application. The *macro* lifetime of a block is the relation between all ops within its lifetime and the total ops count of the application. A block with a small macro lifetime therefore is an object that has a short life span, whereas a block with a large macro lifetime is an object with a large life span. Typically a large value for macro lifetime means it's a global object and can be modelled as such.

A coarse locking lifetime based on the macro lifetime, with a threshold of 50%, is calculated at memtrace-to-ops translation time, as described in Section 5.2 above. The threshold value 50% is based on the assumption that any object that has more than half of all memory accesses in one iteration of a loop is the primary object on which the loop operates.

Based on the relation between ops accessing the block in question and ops accessing other objects the access pattern of the object can be modeled. For example, if an object has 100 ops within its lifetime, 90 of them its own and 10 others, chances are it's an inner loop where the object is locked from its creation to its destruction. Similarly, an object that is accessed few times during its lifetime, compared to all others, would be locked as-needed. Calculating lifetime requires a full opsfile, including all access ops.

It turns out that for some (larger) applications, lifetimes of short-lived objects are highly clustered, as seen in Figure 5.1. This is calculated by the tool `translate-ops-to-histogram.py` as described in Section 5.4 below and visualised here.

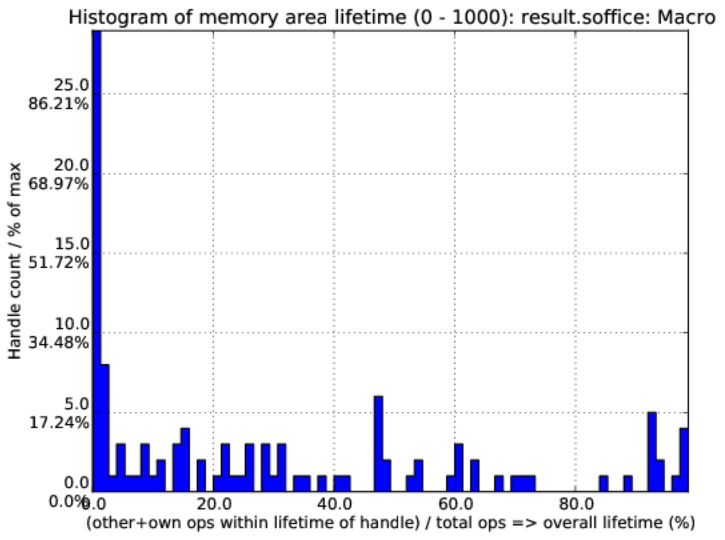


Figure 5.1: This shows the number of objects within a specific lifetime. Short-lived objects dominate.

By removing the short-lived objects, we can get a better understanding of the distribution of the other objects in Figure 5.2.

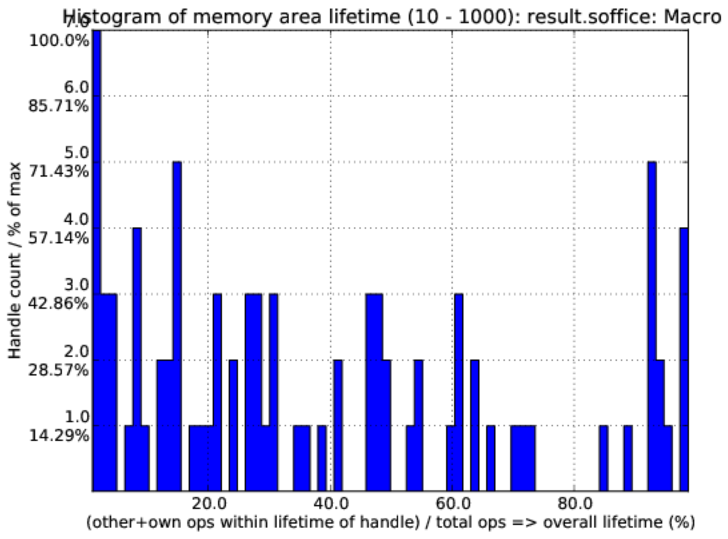


Figure 5.2: Limited to blocks with a lifetime between 1% and 100%

And conversely, if we want to see the distribution of the short-lived objects only, as in Figure 5.3.

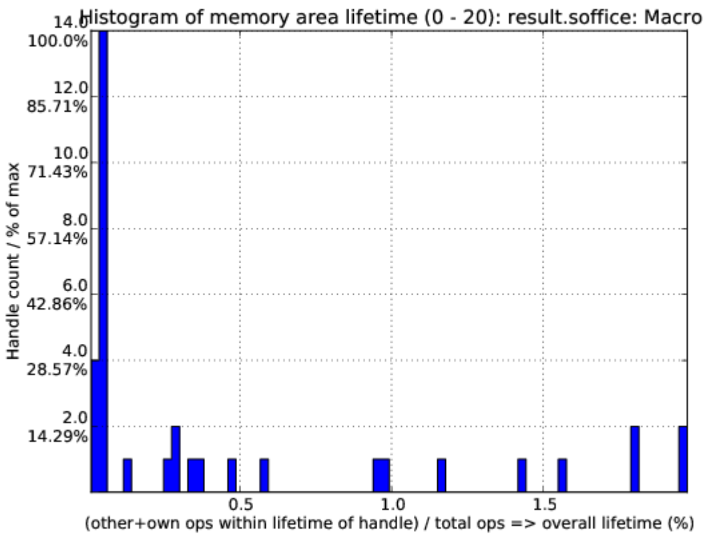


Figure 5.3: Limited to blocks with a lifetime between 0% and 2%

5.4 Lifetime Calculation

Coarsely grained lifetime calculation is done automatically when the raw memtrace is translated into ops, as described above in Section 5.2. The method I'll describe in the following section is more refined but takes more time to calculate. Like the coarse calculation, it is also automatic. All steps from measuring memory access patterns, through simulating allocator performance for that specific app, down to creating graphs displaying memory and speed performance, are automatic.

The algorithm works as follows: when a block is initially created, a threshold value (*life*), is set to zero and will either increase or decrease depending on the operations that come between the new operation and the free operation. A memory access op for the current block increases life by 1, and conversely, another block's operation (regardless of type) decreases life by 0.5. Life is not capped in the upper range but has a lower limit of 0. When life is higher than 0, the current operation's lock status is set, otherwise cleared.

The value was chosen by testing different input parameters against random data, and the graphs that looked best were verified against the smaller application memtraces. This is the algorithm used, with different values for percent, float speed and sink speed:

```
let life = 0
let lifetime = empty array
let number of points = 1000
for i from 0 to number of points:
```

```
let operation belongs to current handle = random() < percent
if operation belongs to current handle:
    life = life + float_speed
else:
    if life >= sink_speed:
        life = life - sink_speed

lifetime.append(life)
```

The results are shown in Figure 5.4.

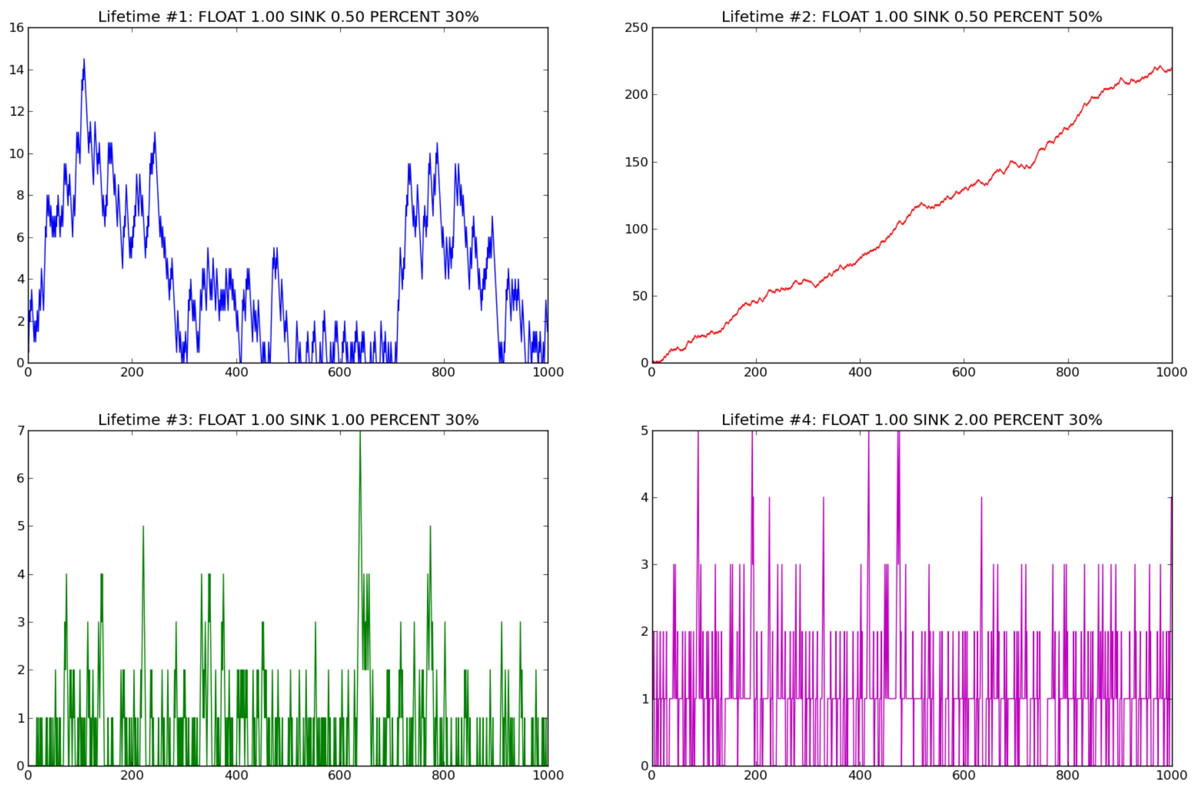


Figure 5.4: Simulated lifetime calculations by varying the values of input parameters.

Clockwise from upper left corner, we see that lock status (i.e. life > 0) varies if the current handle is less than 30% of the ops, and if it's less than 50%, it'll diverge towards always being locked -- which is sound, since any object that is accessed so often is likely to be locked during its lifetime. With sink equal to or larger than float, a jagged graph is produced where the current object is constantly locked/unlocked. A real-world application would want to lock the object once per tight loop and keep it locked until done, instead of continuously locking/unlocking the handle inside the loop. The time under the graph where lifetime is non-zero is one iteration of the loop.

When all ops have been processed, they are written out to a new file that in addition to the regular ops also contains detailed locking information. Since the number of objects is large and the calculation is independent of other objects, the process can be broken down into smaller tasks. This is done using the Python `multiprocessing` module. By recording start and stop indices (based on the New or Free ops, respectively) into the input list, the list of start indices can be broken down into smaller parts to maximize usage of multi-core systems. To saturate the CPU, the tools automatically pick the number of cores plus two as the number of worker threads.

In the case of no corresponding Free operation for the block, no lifetime calculation is done, i.e. it is assumed to be unlocked. This is a limitation of the calculation based on the observation of applications that have a large amount of objects that are never explicitly freed. An implicit free could be inserted at the point of the last memory access, but it is not implemented.

The fine grained calculation of this method is slower ($O(m*n)$, where m is the number of handles and n is the total number of operations), but intersperses lock/unlock instructions throughout the lifetime of an object, instead of forcing the object to be locked its entire lifetime. The more fine-grained locking/unlocking, specifically unlocking, the more efficiently compacting can be performed.

Hand tuning an application with lock/unlock inserted at the most appropriate locations as determined by manual static analysis and knowledge of the application and comparing it to the approximated lifetime calculation, is not done in this report, and would be a good subject for future work.

Chapter 6

Jeff: The Compacting Allocator

6.1 Overview

In order to achieve compacting, memory must be accessed indirectly. This is the signature:

```
void rminit(void *heap, size_t heap_size);
handle_t rmmalloc(size_t nbytes);
void rmfree(handle_t *handle);
void *lock(handle_t handle);
void unlock(handle_t handle);

void compact(uint32_t compact_time_max_ms);
```

`handle_t` is an opaque type. To get the actual memory pointed to at by the handle, call *lock()* on it to obtain a normal pointer to memory. During the time a block pointed out by a handle is locked, the compact operation is not allowed to move it. If it could be moved, the pointer obtained by the client code would no longer be valid. This also puts certain limitations on the compactor, since it needs to deal with possibly locked blocks. Client code needs to be adapted to this allocator, such that memory is always appropriately locked/unlocked as needed. The compacting operation is discussed in more detail in Section 6.2.4.

6.2 Implementation

In the previous sections I described the general functionality. This section will in more details describe how each of the key parts work, including memory layout and performance metrics.

6.2.1 rminit

Recall the signature:

```
void rminit(void *heap, size_t heap_size);
```

heap is the client-supplied heap of size heap_size. Jeff is self-contained within the heap and requires no additional storage except for stack space.

Internal structures are initialized:

- Boundaries (g_memory_bottom, g_memory_top)
- Header blocks (g_header_root, g_unused_header_root)
- Free block slots (g_free_block_slots)

I'll go through each one of them below, and their uses will be clarified as I touch upon them later in the other parts of the allocator.

Boundaries (g_memory_bottom/g_memory_top)

Bottom of memory is the bottom of the heap and top is the highest used memory address. Compacting resets the top to the highest used memory address.

Header blocks (g_header_root and g_unused_header_root)

Linked lists of all headers and the root of all unused headers.

The opaque type handle_t is a pointer to a header_t structure:

```
typedef struct header_t {
    void *memory;
    uint32_t size;
    uint8_t flags;

    struct header_t *next;
#if JEFF_MAX_RAM_VS_SLOWER_MALLOC == 0
    struct header_t *next_unused;
#endif
};
```

This is the minimum amount of memory used by a block. Assuming a 32-bit system, memory is 4 bytes, size is 4 bytes and flags is 1 byte. The header itself is a linked list (next) that can be sorted in memory order in the compact step, since the handles themselves cannot be moved as they're used (in disguise) by the client code. Flags can have one of the following values:

- Free (0)
- Unlocked (1)
- Locked (2)
- Weakly locked (4) (not implemented)

A weakly locked block can be treated as unlocked in the compacting phase so it can be reclaimed. Care needs to be taken by the client code since compacting invalidates the pointer to memory.

The array of header items grows down from the top of the client-supplied heap. New handles are searched for starting at g_header_top and down until g_header_bottom. If there is no free header when requested and there is no overlap between existing memory (including the newly requested size in case of a malloc), `g_header_bottom` is decreased and a fresh handle is returned. If `g_header_bottom` and `g_memory_top` are the same, `NULL` is returned to signal an error.

The optional member `next_unused` is a compile-time optimization for speeding up the *O(n)* find header operation to *O(1)* at the expense of memory. `g_unused_header_root` is set to the header that is newly marked as unused and the next pointer is set to the old unused header root. Setting `memory` to `NULL` indicates an unused header.

`g_header_root` points to the latest used header. At compact time, it's sorted in memory order.

Free block slots (g_free_block_slots)

As touched upon earlier, this contains the memory blocks that have been freed and not yet merged into unused space by a compact operation:

```
typedef struct free_memory_block_t {
    header_t *header;
    struct free_memory_block_t *next; // null if no next block.
} free_memory_block_t;
```

When a block is freed, a `free_memory_block_t` is stored in the first bytes. Therefore, the minimum block size is 8 bytes, assuming a 32-bit system. The `header` field stores the actual information about the block. By checking `header->memory` against the address of the `free_memory_block_t` instance, we know if it's a valid free memory block. The `next` field points to the next block in the same size range.

There are $log_2(heap\_size)$ (rounded up) slots. Freeing a block of size 472 bytes means placing it at the start of the linked list at index 9 and hanging the previous list off the new block's next pointer, i.e. a stack, and is rebuilt at compact time. Adding a free block takes constant time.

6.2.2 rmmalloc

Minimum allocatable size is `sizeof(free_memory_block_t)` for keeping information about the block for the free list. I'll go through the process of allocation step by step.

There are two cases: either there is space left after top of the memory for a header and the requested memory, in which case the fast path is taken where a header is allocated, `g_memory_top` is bumped and the header is associated with the newly created memory and returned to the client. Allocating a header means searching the header array for an unused block, or if the optimization described above is enabled, following `g_unused_header_root`. If no header is found, `g_header_bottom` grows downward if there is space, but there are always two headers left for compacting (more on that in the section on compacting).

In the other case, there is no space left after `g_memory_top` and the free block list must be scanned for an appropriate block. This is the most complex part of alloc/free.

The time complexity of the first case with the aforementioned optimization is *O(1)*, or *O(n)* (in terms of number of handles in the system) in the unoptimized case. In the second case where memory can't grow up, the time complexity is worst case *O(n)* (in terms of the number of blocks of the specific size) and best case *O(1)*.

Find free block

Calculate the index $k = log_2(size)+1$ into the free block slots list. As explained earlier, the free block slot list has a stack (implemented as a singly linked list) hanging off each slot, such that finding a suiting block will be a fast operation. The exeption is for requests of blocks in the highest slot have to be searched in full, since the first block found is not guaranteed to fit the size request, as the slot k stores free blocks $2^{k-1} \leq n < 2^k$ and there is no larger $k+1$ slot to search in.

In the normal case the free block list is looked up at k for a suiting block. If the stack is empty, k is increased and the free block list again is checked until a block is found. Finally, if there was no block found, the actual index $log_2(size)$ is searched for a block that will fit. Remember that the blocks in a specific slot can be $2^{k-1} \leq n < 2^k$ and therefore there could be free blocks in slot k that are large enough for the request. When a block is found, it's shrunk into two smaller blocks if large enough, one of the requested size and the remainder. Minimum required size for a block to be shrunk is having one extra header available and that the found block is `sizeof(free_memory_block_t)` bytes larger than the requested size. Otherwise, the block is used as-is causing a small amount of internal fragmentation. The remainder of the shrunk block is then inserted into the tree at the proper location.

Returns `NULL` if no block was found.

Shrink block

Adjusts size of current block, allocates a new header for the remainder and associates it with a `free_memory_block_t` and stores it in the shrunk block.

6.2.3 rmfree

Mark the block as unused.

6.2.4 rmcompact

The compacting operation consists of setup, compacting and finish.

Start with sorting all memory headers by pointer address, such that `g_root_header` points to the lowest address in memory and by following the `next` pointer until `NULL` all blocks can be iterated. All blocks have a header associated with them, regardless of flags. This step only has to be done once each call to `rmcompact()`.

Actual compacting is done in passes so it can be optionally time limited, with a granularity of the time it takes to perform a single pass, so it is not a hard limit. Also, the sorting in the beginning and the free block list rebuilding in the end is not included in the time constraint.

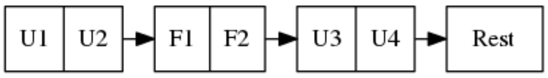
One pass of moving blocks around

1. Get closest range of free headers (or stop if no headers found)

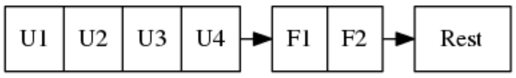
    1. If block directly after free header is locked, set a max size on unlocked blocks.

2. Get closest range of unlocked headers (respecting max size if set)
    1. No blocks found and limitation set on max size: if free blocks were passed searching for unlocked blocks, try again from the block directly after the free headers, else stop.
    2. Set adjacent flag if last free's next is first unlocked
3. Calculate offset from free area to unlocked area
4. Squish free headers into one header and associate memory with the header
5. Move unlocked blocks to free area
    1. Move data
    2. Adjust used header pointers
6. Adjacent: relink blocks so unlocked headers are placed before what's left of free area, and free area pointing to header directly following previous position of last unlocked header's next header:

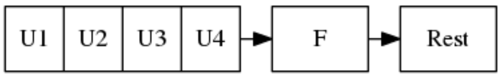
   Initial configuration with blocks Unlocked 1-4, Free 1-2, Rest:



   Move all used blocks back (i.e. to the left), relink free blocks:

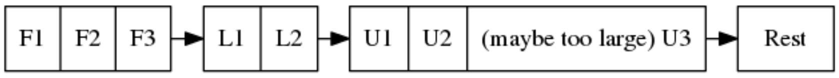


   Squish free blocks:

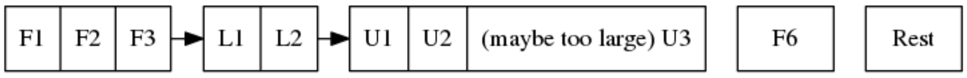


7. Non-adjacent: similar to adjacent, except blocks can't just be simply memmov'ed because of the locked blocks. Instead, only the blocks that fit in the free space can be moved:

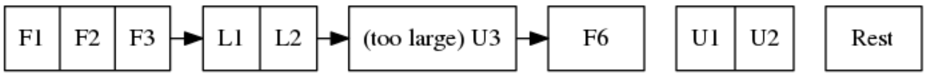
   Initial configuration with blocks Free 1-3, Locked 1-2, Unlocked 1-3, Rest:



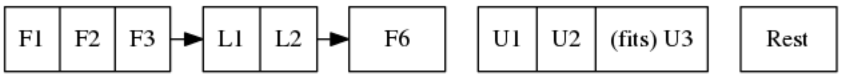
   Create free block 6 in the area where the used blocks are now:



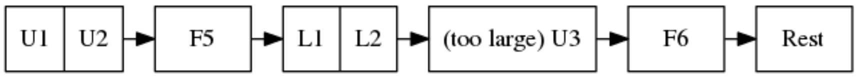
   Either: a) Block U3 is too large to fit in the free area:



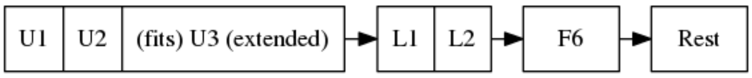
Or: b) Block U3 fits in the free area.



Then, Either: a) With a new block Free 5 with left-overs from Free 1-3 and F6 from the space between U1-U3 and Rest:



Or: b) Unlocked 3 fits, but not enough size to create a full block F5 -- instead extend size of Unlocked 3 with 0 < n < sizeof(free_memory_block_t) bytes:



8. Continue to next round, repeating until time limit reached or done (if no time limit set)

Finishing

At the end of the compacting, after the time-limited iterations, finishing calculations are done: calculate the highest used address and mark all (free) headers above that as unused, adjust `g_header_bottom` and finally rebuild the free block slots by iterating through `g_header_root` and placing free blocks in their designated slots.

6.2.5 rmdestroy

Doesn't do anything - client code owns the heap passed to `rminit()`.

6.3 Testing

As described in Chapter 3, unit testing is utilized where applicable.

6.3.1 Real-World Testing

Since the allocator does not have the interface of standard allocators client code needs to be rewritten. The two major problems with this is that it requires access to source code, and rewriting much of the source code. This is where Steve (Chapter 7) is useful.

6.4 Profiling

The GNU profiling tool *gprof*[14] was used to find code hotspots, where the two biggest finds were:

- `log2()`
- `header_find_free()`

In the spirit of first getting things to work, then optimize, the original `log2()` implementation was a naive bitshift loop. Fortunately, there's a GCC extension `__builtin_clz()` (Count Leading Zeroes) that is translated into efficient machine code on at least x86 that can be used to write a fast `log2(n)` as `sizeof(n)*8 - 1 - clz(n)`. The hotspots in the rest of the code were evenly distributed and no single point was more CPU-intense than another, except in `header_find_free()`. As described above, there's a compile-time optimization that cuts down time from *O(n)* to *O(1)*, which helped cut down execution time even more at the expense of higher memory usage per block.

More details and benchmarks in Chapter 7.

6.5 Automatic Testing

I've introduced bugs in the functions called from the allocator interface to see if the testing framework would pick them up. The idea is to introduce small changes, so-called *off-by-one errors*, where (as the name suggest) a value or code path is changed only slightly but causes errors. Below is a list of example bugs that the automatic tests found, and could later be fixed. Automatic testing is useful.

Function `free_memory_block_t *block_from_header(header_t *header)`:

```
return (free_memory_block_t *)((uint8_t *)header->memory +
header->size) - 1;
```

Fuzzed

```
return (free_memory_block_t *)((uint8_t *)header->memory +
header->size);
```

Function `uint32_t log2_(uint32_t n)`:

```
return sizeof(n)*8 - 1 - __builtin_clz(n);
```

Fuzzed

```
return sizeof(n)*8 - __builtin_clz(n);
```

Function `inline bool header_is_unused(header_t *header)`:

```
    return header && header->memory == NULL;
```

Fuzzed

```
    return header && header->memory != NULL;
```

Function `inline void header_clear(header_t *h)`:

```
    h->memory = NULL;
    h->next = NULL;
```

Fuzzed (1)

```
    //h->memory = NULL;
    h->next = NULL;
```

Fuzzed (2)

```
    h->memory = NULL;
    //h->next = NULL;
```

Function `header_t *header_new(bool insert_in_list, bool spare_two_for_compact)`

```
    ...
    header->flags = HEADER_UNLOCKED;
    header->memory = NULL;
    ...
    if ((header->next < g_header_bottom || header->next >
    g_header_top) && header != g_header_root) {
    ...
```

Fuzzed (1)

```
    ...
    //header->flags = HEADER_UNLOCKED;
    header->memory = NULL;
    ...
    if ((header->next < g_header_bottom || header->next >
    g_header_top) && header != g_header_root) {
    ...
```

Fuzzed (2)

```
    ...
    header->flags = HEADER_UNLOCKED;
    header->memory = NULL;
    ...
    if ((header->next < g_header_bottom || header->next >
    g_header_top)) {
    ...
```

Function `header_t *block_free(header_t *header)`

```
    block->next = g_free_block_slots[index];
    g_free_block_slots[index] = block;
```

Fuzzed

```
    g_free_block_slots[index] = block;
    block->next = g_free_block_slots[index];
```

Function `free_memory_block_t *freeblock_shrink_with_header(free_memory_block_t, header_t *, uint32_t)`

```
    h = header_new(/*insert_in_list*/true, /*force*/false);
```

Fuzzed (1)

```
    h = header_new(/*insert_in_list*/false, /*force*/false);
```

Fuzzed (2)

```
    h = header_new(/*insert_in_list*/true, /*force*/true);
```

Function `header_t *freeblock_find(uint32_t size)`

```
    int target_k = log2_(size)+1;
```

Fuzzed

```
    int target_k = log2_(size);
```

Function `rmcompact(int maxtime)`

```
    uint32_t used_offset = header_memory_offset(free_first,
    unlocked_first);
    ...
    header_t *free_memory = header_new(/*insert_in_list*/false,
    /*force*/true)
```

Fuzzed (1)

```
uint32_t used_offset = header_memory_offset(free_first,
free_last);
```

Fuzzed (2)

```
header_t *free_memory = header_new(/*insert_in_list*/true,
/*force*/true)
```

[14] http://www.gnu.org/software/binutils/

Chapter 7

Steve: The Benchmark Tool

Steve is a benchmark tool for collecting and visualizing runtime memory access and allocation patterns in arbitrary applications that use the system malloc, without access to source code, and performing tests without running the actual application.

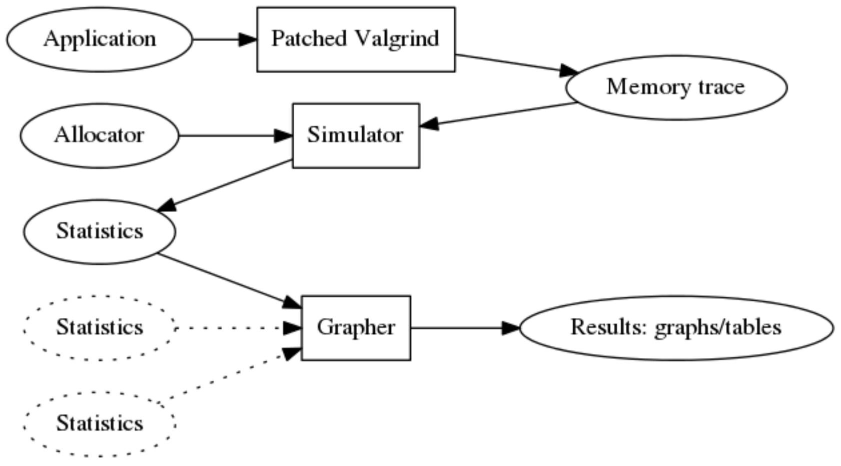


Figure 7.1: Architectural diagram of Steve

Multiple statistics files can be fed to the grapher for doing comparisons between different allocators on the same input data set.

7.1 Modules

Steve is a collection of modules:

- `translate-memtrace-to-ops.py`
- `translate-ops-to-histogram.py`
- `translate-ops-to-locking-lifetime.py`

- `run_allocator_stats.sh`, `run_allocator_stats_payload.sh`
- `run_memory_frag_animation.sh`
- `run_graphs_from_allocstats.py`
- `run_memory_frag_animation_plot_animation.py`

For a detailed description, see the appendix.

7.2 Allocator Driver Usage

Steve does, in essence, two tasks: visualize memory and plot benchmark data. The framework allows for easy extension with more tools.

- `run_memory_frag_animation.sh`: create an animated memory allocation visualisation.
- `run_graphs_from_allocstats.py`: create benchmarks based on one or many allocator statistics inputs (generated by `run_allocator_stats.sh`)

The tools are described in more detail in the next section.

All alloc drivers are linked to the same main program and have the same command line parameters:

- `--peakmem opsfile`

  Prints out theoretical heap size allocated as reported by the allocator driver. `--allocstats` passes this data to benchmark data files for later processing by the graphing tool.

  Parameters:

  – opsfile - operations file created by `translate-memtrace-to-ops.py`.

- `--allocstats opsfile resultfile killpercent oplimit peakmemsize theoretical_heap_size`

  Generates a file in JSON format in the following format. Header:

```
driver = "jemalloc"
opsfile = "result.program-ops"
heap_size = 13544700
theoretical_heap_size = 4514900
opmode = 'allocstats'
alloc_stats = [
```

  Then, per line a dictionary with the following keys:

```
{'op_index':        <sequene number>,
 'free':            <bytes: integer>,
 'used':            <bytes: integer>,
 'overhead':        <bytes: integer>,
 'maxmem':          <bytes: integer>,
 'current_op_time': <microseconds: integer>,
 'oom_time':        <microsecond: integer>,
 'optime_maxmem':   <microsecond: integer>,
     'op':               <operation <- N, F, A, L, U: char>,
     'size':             <bytes: integer>
    }
```

Parameters:

- opsfile: Operations file created by `translate-memtrace-to-ops.py`.
- resultfile: Statistics output file, convention is to use file stem of opsfile (without "-ops") and append "-allocstats"
- killpercent: Optionally rewind and randomly free *killpercent* (0-100) of all headers at EOF (end-of-file), to simulate an application that destroys and creates new documents. The value 100000 means no rewinding or killing takes place, i.e. just one round of the data gathered by running the application to be benchmarked.
- oplimit: Which operation ID (*0..total ops count*) to write allocation stats for. The special value 0 is for writing the original header. Typically the driver application is called in a for loop from 0 to the number of operations, i.e. number of lines in the opsfile.

• `--memplot opsfile [heap_size]`

For each operation, call out `run_memory_frag_animation_plot_animation.py` to create a PNG of the heap at that point in time. The driver application only needs to be run once.

Parameters:

- opsfile - operations file created by `translate-memtrace-to-ops.py`.
- (optional) heap_size - maximum heap size to use

These are not called directly, but instead called from by the modules described below.

At startup the mode of operation of the allocator driver is set to one of these. All modes perform follow the same basic flow:

1. Allocate heap according to specified heap size or use predefined size (currently 1 GB). If heap allocation fails, decrease by 10% until success.
2. Allocate and initialize colormap as $\frac{1}{4}$ of heap size. (more on colormap later)
3. Initialize driver.
4. Initialize randomness with compile-time set seed.
5. Open opsfile.
6. Run mode's main loop.
7. Save statistics created by mode's main loop.
8. Destroy driver.

The main loop follows the same basic structure:

1. Scan a line of the ops file and put in the variables handle, *op*, *address* and *size*.
2. Switch on op:
    - Op is N (New): Call `user_malloc` with the size. On OOM, call `user_handle_oom` and call `user_malloc` again if successfully handled. Make sure that there was no OOM on the final malloc. Retrieve the highest address in use by `user_highest_address()`. Store object pointer (that may or may not be a directly accessible memory address) and memory address (if available) from malloc along with size in hash tables keyed on the handle id.
    - Op is F (Free): Retrieve the object pointer and call `user_free`.
    - Op is L (Lock): Retrieve the object pointer and all `user_lock`.
    - Op is U (Unlock): Retrieve the object pointer and all `user_unlock`.

    Access (load, store, modify) operations are not handled in the loop since their use is limited to calculating lifetime statistics and locking behaviour.
3. Exit on EOF.

7.3 Driver Modes

In this section, I'll describe the specifics on the three main loops (peakmem, allocstats, memplot) and then the tools that use them.

7.3.1 peakmem

Find the largest amount of memory during the driver's lifetime for a specific opsfile, as calculated by the highest address+size of a block minus the start address of the heap. This number is used as a theoretical maximum heap size to measure the amount of overhead incurred by the allocator, used by the module `run_allocator_stats.sh`.

7.3.2 allocstats

The purpose is to allow for the driver application to run several rounds of the application data, as explained above, to do a rough simulation of an application creating and destroying documents. It stores timing info for `new` and `free` and adds rewinding of the input file and random free of a certain percentage, if requested, of the allocated objects on opsfile EOF.

Used by the module `run_allocator_stats.sh`.

7.3.3 memplot

At each operation, a *colormap* is updated with all known objects. In order to retrieve the physical memory address they are locked (throuh `user_lock`) and the pointer is registered.

Colormap is $\frac{1}{4}$ the size of the heap size, such that each 4-byte word maps onto a byte. The colormap is initially filled with white (for overhead), with a `new` operation painted as red and `free` painted as green. The heap is correspondingly filled with `HEAP_INITIAL` (`0xDEADBEEF`) initially, and newly created blocks are filled with `HEAP_ALLOC` (`0xBEEFBABE`) and blocks that are just about to be freed are filled with `HEAP_FREE` (`0xDEADBABE`).

Now, by scanning the heap for values that are not in the set `HEAP_INITIAL`, `HEAP_ALLOC` nor `HEAP_FREE`, it can be concluded that this is overhead (i.e. allocator-internal structures). Paint the corresponding memory location in the colormap with white (for overhead). It also adds non-optional rewinding to run until OOM.

7.4 Tested Allocators

The allocator often used by Linux and elsewhere in the open-source world is Doug Lea's Malloc *dlmalloc*, that performs well in the average case. For FreeBSD, Poul-Henning Kamp wrote an allocator that he aptly named *pkhmalloc*. *dlmalloc* aims to be good enough for most single-threaded use cases and is well-documented, therefore attractive to anyone in need of an allocator. It does not perform optimally in multi-threaded applications because of the coarse (operation-level) locking. Other allocators are designed to be used in a mutli-threaded application where locking is performed on a finer level, not blocking other threads trying to use the allocator at the same time.

In fact, at Opera, *dlmalloc* was used internally to better tune allocator characteristics for memory-constrained devices, where all available memory was requested at startup and then used by the internal malloc.

All allocators' `new` and `free` calls mapped to the corresponding function call. Handle OOM is a no-op except for Jeff. No allocators use `mmap`.

7.4.1 rmmalloc (Jeff)

Maps all `user_...` calls to the corresponding calls in Jeff. For the compacting version, `user_handle_oom` always performs a full compact, and on the non-compacting version, `user_handle_oom` is a no-op.

The workings of Jeff is described earlier in this paper.

7.4.2 jemalloc (v1.162 2008/02/06)

jemalloc is an allocator written by Jason Evans, originally written for a custom development environment circa 2005, later integrated into FreeBSD for its multi-threading capabilities and later further adapted in 2007 for use by the Firefox project to deal with fragmentation issues. It's since been adapted for heavy-duty use in the Facebook servers[15]. As of 2010, it still performs better than the system-provided allocators in MacOS, Windows and Linux.[16]

[15]`https://github.com/jemalloc/jemalloc/wiki/History`

[16]`http://www.quora.com/Who-wrote-jemalloc-and-what-motivated-its-creation-and-implementation`

7.4.3 dlmalloc v2.8.6

dlmalloc is an allocator written by Doug Lea and is used by the GNU standard C library, glibc. The source code states the following about its goal:

> This is not the fastest, most space-conserving, most portable, or most tunable malloc ever written. However it is among the fastest while also being among the most space-conserving, portable and tunable. Consistent balance across these factors results in a good general-purpose allocator for malloc-intensive programs.

7.4.4 tcmalloc (gperftools-2.1)

gperftools[17] is written by Google and includes a profiling/benchmark framework/tools. It is used by, among others, Google Chrome, MySQL and WebKit Fang (2012), which in turn is used by many other projects such as Apple's Safari. It includes the allocator *tcmalloc*.

[17]`http://code.google.com/p/gperftools/`

Chapter 8

Results

8.1 Limitations

Both tcmalloc and jemalloc perform poorly with `mmap()` disabled, and in some cases they did not manage to finish allocation simulation. In those cases the allocators are omitted from the results table. The first thing Steve does is to calculate the maximum heap size used by the allocator for an application, by starting from the theoretical heap size for an ideal allocator and increasing that value until no OOMs occur. If the limit on that increase is reached, the allocator is marked as *did not finish*.

8.2 Input Data

Measuring an allocator must be done in conjunction with input data. These are the applications tested

- Opera v12.0[18]: load `http://www.google.com` and exit.
- StarOffice (LibreOffice) 4.0.2.2[19]: open a blank word processor document and exit.
- sqlite 2.8.17 (Ubuntu 13.04's default version)[20]: load 17 MB phpBB3[21] SQL data.
- ls 8.20 (Ubuntu 13.04's default version)[22]: display the `/bin` directory.
- latex 3.1415926-2.4-1.40.13[23]: paper.tex (96 lines, 2.6 KB).
- GNU tar 1.27.1 (Ubuntu 13.04's default version)[24]: compressing the contents of the Valgrind 3.9.0 source distribution (87 MB).

The results are presented in charts and tables. I'll describe what they mean first, then give the results.

8.3 Keys To Read Charts and Tables

8.3.1 Drivers

- **rmmalloc**: Jeff without compacting
- **rmmalloc-c**: Jeff with compacting
- **rmmalloc-c-m**: Jeff with compacting and maximum memory tweak

8.3.2 Charts

There are two types of charts, one of performance in time and one of performance in space.

Speed chart.

- **X axis**: A counter that is increased by one at each new, free, lock and unlock operation.
- **Y axis**: The execution time of the operation, on a *log10*-scale.

Size chart.

- **X axis**: Same as above.
- **Y axis**: The maximum allocatable amount of memory relative to the maximum heap size at each point in time, by running the application to that point, trying a maximum allocation and then restarting the application, continuing to to the next point.

8.3.3 Tables

Scoring explained:

- Let $A_1..A_n$ be all allocators.
- Let $O_1..O_m$ be all operations in the application currently being measured.
- Let S_{ij} where $i = 1..n, j = 1..m$ be the score for the allocator i at operation j, such that $S_{ij} \in 0..n$, where 0 is the best result and n is the worst.
- Let $P_i \in 0..1.0$ be the penalty of any allocator $i \in A_1..A_n$, defined as $P_i = \frac{1}{n*m}\sum_{j=1}^{m} S_{ij}$, where 0 is best and 1.0 is worst.

[18] `http://www.opera.com`

[19] `http://www.libreoffice.org`

[20] `http://www.sqlite.org`

[21] `http://www.phpbb.com` - a bulletin-board system

[22] `http://www.gnu.org/software/coreutils/`

[23] `http://www.latex-project.org`

[24] `https://www.gnu.org/software/tar/`

- Let $B_i \in A_1..A_n$ be the number of times the allocator i has performed best.
- Let $W_i \in A_1..A_n$ be the number of times the allocator i has performed worst.

Space table, sorted in descending order with best first.

- **Driver**: Name of the driver
- **Penalty (c)**: As given above.
- **Penalty (w)**: Score weighted by the distance to the lowest scoring allocator. Let b be the best performing allocator, then $Sw_{ij} = \frac{S_{ij} - S_{bj}}{S_{bj}}$ where b is the best performing allocator.
- **Best**: Ratio of $\frac{B_i}{n} \in 0..1.0$.
- **Worst**: Ratio of $\frac{W_i}{n} \in 0..1.0$.

Speed table, same sorting as the space table. In addition to the fields in speed table (applied to size, instead of speed), these fields are defined:

- **Average**: Average speed of all operations for a given allocator.
- **Median**: Median speed of all operations for a given allocator.

Penalty (c) can be considered to be the average internal ranking of an allocator, whereas penalty (w) shows the average internal weighted by the distance to the best allocator. Therefore, penalty (w) gives the reader a clue on the allocator's absolute performance, and it is also less smoothed out by simply averaging. An example of this can be seen in Table 8.1 below.

All tables are sorted by penalty (c).

8.4 Results

The results are very interesting in that there's a variation between the allocators, which of course is expected, but also between the different applications tested, each with a unique memory usage patterns. Two separate patterns can be discerned when it comes to speed, with Figures 8.1, 8.2, 8.3 in one group and 8.4, 8.5, 8.6 in the other.

Something else to note is that jemalloc performs badly, very likely because of the limitation to only use *sbrk()* for requesting memory from the operating system.

8.4.1 StarOffice

Command line used: `soffice`

Simulated using full lockops.

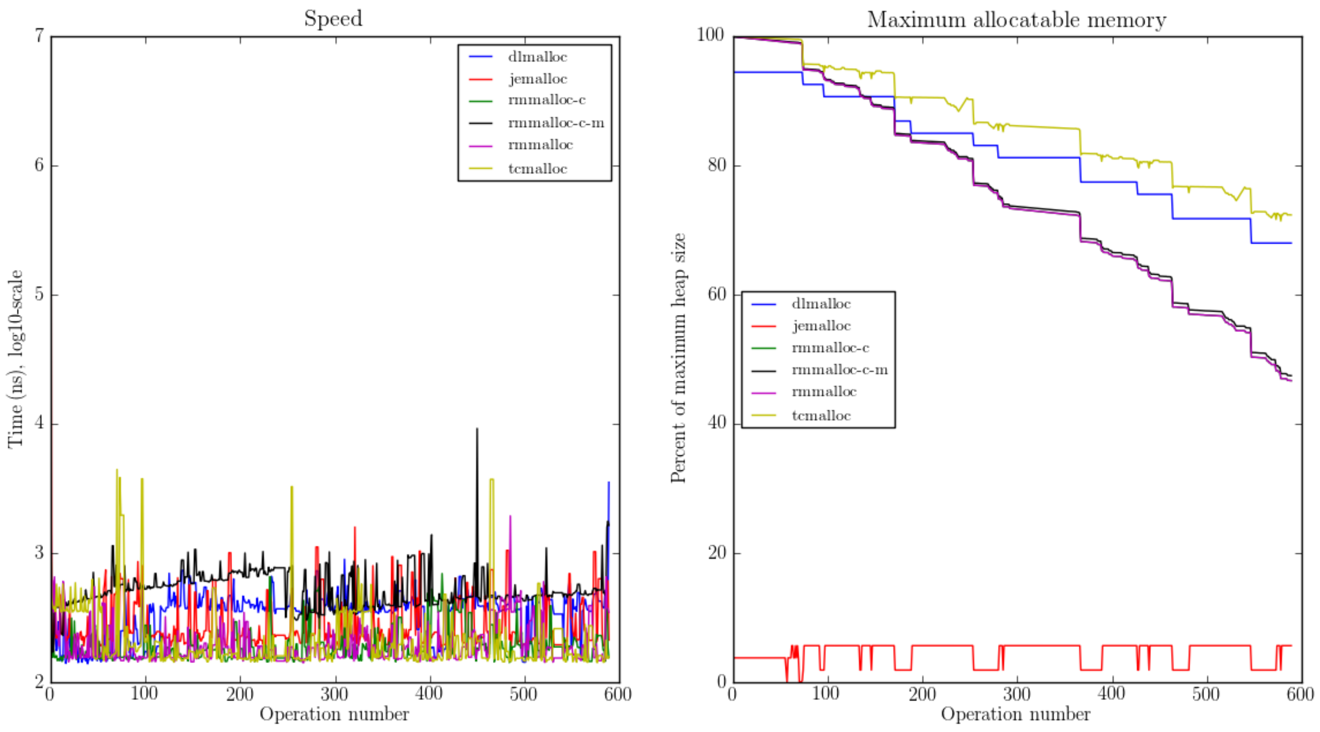


Figure 8.1: Soffice results. Poor performance of jemalloc.

The chart in Figure 8.1 clearly shows the space performance of the tested allocators, whereas the speed chart is harder to read because of the similar speeds and the number of allocators tested. Tables 8.1 and 8.2 are particularly useful here.

Speed					
Driver	**Penalty (c/w)**	**Best**	**Worst**	**Average**	**Median**
rmmalloc	23% / 18.30%	30.00%	3.22%	209 ns	171 ns
rmmalloc-c	23% / 15.80%	27.29%	1.69%	205 ns	178 ns
tcmalloc	25% / 54.76%	34.07%	6.44%	286 ns	164 ns
jemalloc	47% / 1378.68%	0.34%	10.00%	9751 ns	228 ns
dlmalloc	54% / 87.60%	8.14%	11.86%	372 ns	370 ns
rmmalloc-c-m	75% / 205.50%	0.17%	66.78%	562 ns	483 ns

Table 8.1: Speed measurements for soffice

Space			
Driver	**Penalty (c/w)**	**Best**	**Worst**
tcmalloc	0% / 0.00%	100.00%	0.00%
dlmalloc	28% / 1.42%	0.00%	0.00%
rmmalloc-c-m	29% / 4.36%	0.00%	0.00%
rmmalloc	46% / 6.80%	0.00%	0.00%
rmmalloc-c	62% / 9.08%	0.00%	0.00%
jemalloc	83% / 78.88%	0.00%	100.00%

Table 8.2: Space measurements for soffice

As explained above, the penalty number by itself can be misleading. For example, in Table 8.1 we see that e.g. both *rmmalloc* and *rmmalloc-c* have the same penalty (c), but they differ in other metrics. Which metric is more important depends on the application at hand. For certain applications, it might be important that it performs predictably, in which case an allocator that has a high best *and* a high worst is a bad choice.

The space metrics in Table 8.2 paints a clearer picture with two outliers, one in *tcmalloc* performing better than all other allocators, and *jemalloc* performing worse than all other allocators.

8.4.2 sqlite

Command line used: `sqlite < gkk_styrkelyft_se.sql`

Simulated using full lockops.

Results in Figure 8.2, Table 8.3 and Table 8.4.

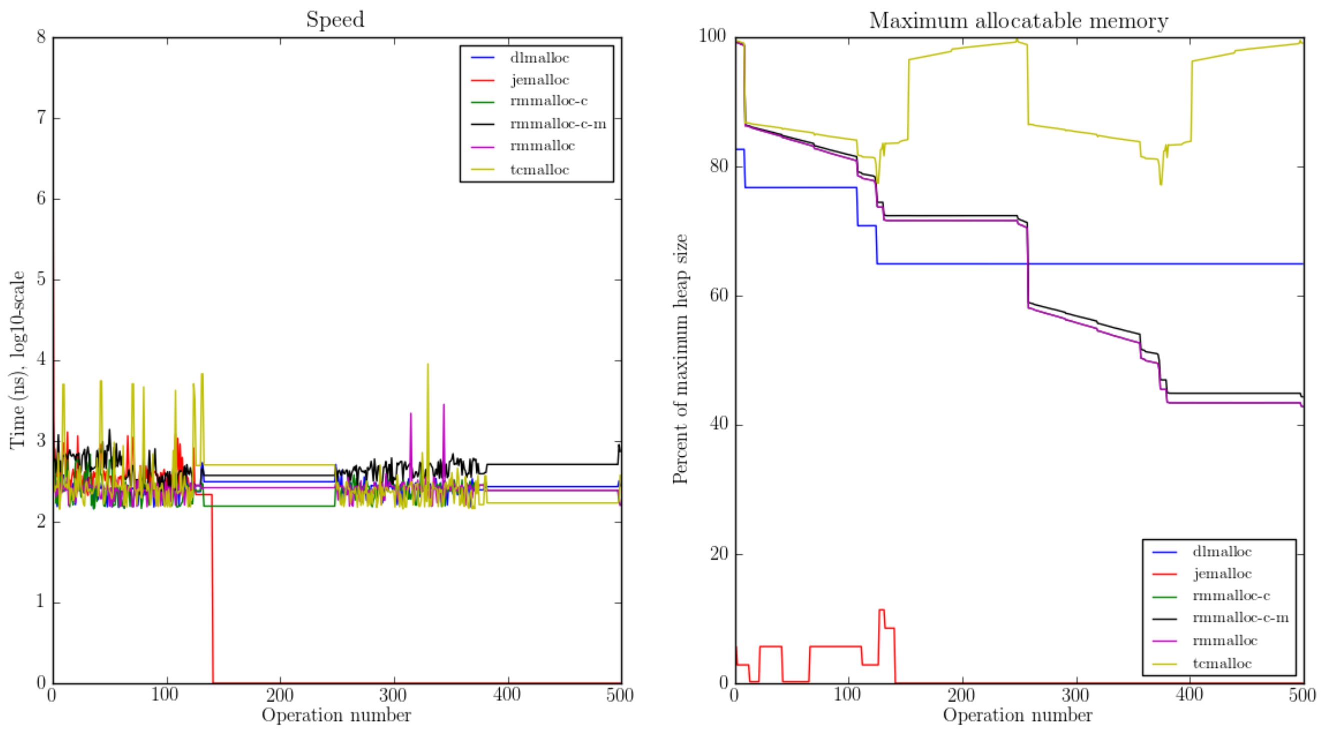


Figure 8.2: Sqlite results.

Again, bad performance of *jemalloc* in both measurements.

Speed					
Driver	**Penalty (c/w)**	**Best**	**Worst**	**Average**	**Median**
jemalloc	14% / 4726.79%	74.00%	5.60%	30152 ns	0 ns
rmmalloc-c	30% / 5718.10%	10.20%	1.40%	236 ns	245 ns
rmmalloc	38% / 8647.63%	6.00%	1.40%	262 ns	257 ns
tcmalloc	42% / 13830.88%	4.80%	28.40%	434 ns	250 ns
dlmalloc	48% / 10978.71%	4.20%	1.00%	286 ns	272 ns
rmmalloc-c-m	75% / 25289.42%	0.80%	62.20%	464 ns	442 ns

Table 8.3: Speed measurements for sqlite

	Space		
Driver	**Penalty (c/w)**	**Best**	**Worst**
tcmalloc	0% / 0.00%	100.00%	0.00%
rmmalloc-c-m	24% / 8.14%	0.00%	0.00%
rmmalloc	41% / 13.22%	0.00%	0.00%
dlmalloc	42% / 9.59%	0.00%	0.00%
rmmalloc-c	58% / 18.02%	0.00%	0.00%
jemalloc	83% / 82.16%	0.00%	100.00%

Table 8.4: Space measurements for sqlite

The speed table 8.3 is slightly confusing with regard to *jemalloc*, but can easily be understood if examined along with the corresponding graph. In fact, it performs rather well, up until the point where it fails to work at all. This also skews the other allocator's penalty (w) numbers, which have to be viewed in relation to the base line which is *jemalloc*. Adjusting the number, we instead get the following penalty (w) numbers:

- jemalloc 100
- rmmalloc-c: 120
- rmmalloc: 183
- tcmalloc: 292
- dlmalloc: 232
- rmmalloc-c-m: 530

As for the space table 8.4, *rmmalloc-c-m* which performed badly in speed instead performs best when it comes to space. A better compromise between the two is *rmmalloc-c*, performing well in both space in time.

8.4.3 ls

Command line used: `ls /bin`

Simulated using full lockops.

Results in Figure 8.3, Table 8.5 and Table 8.6.

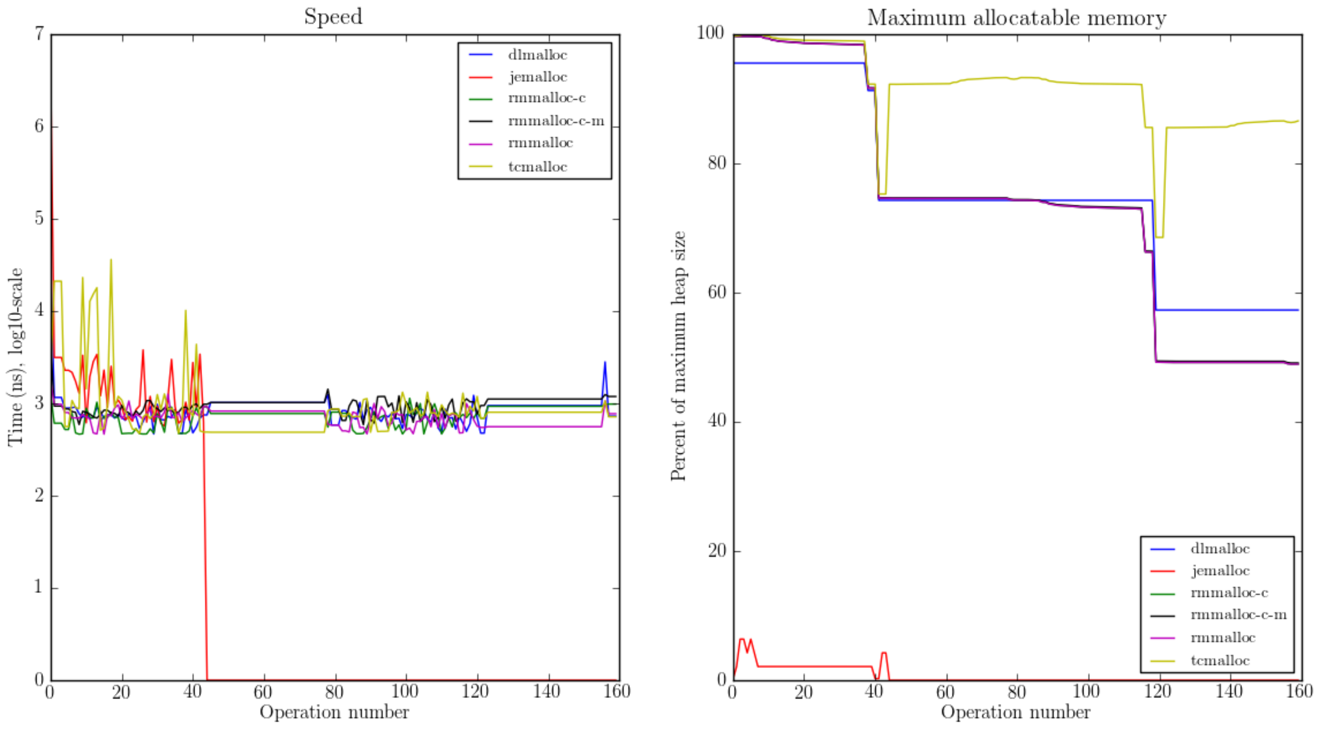


Figure 8.3: ls results.

The memory chart shows how memory allocation and memory use are split up in *ls*. First, it allocates data (op 0-40), then it operates on the data, does more allocation of "simpler" data sizes (a request that takes less time to serve, possibly by being of a size that can be handed out from a small objects pool or similar), followed by more data operations and finally a small allocation operation, most likely a free. Again, *jemalloc* did not survive past the initial allocation operations.

	Speed				
Driver	**Penalty (c/w)**	**Best**	**Worst**	**Average**	**Median**
jemalloc	16% / 1406.11%	73.75%	10.62%	20404 ns	0 ns
rmmalloc-c	35% / 25452.74%	14.38%	3.12%	752 ns	776 ns
rmmalloc	35% / 19559.69%	4.38%	2.50%	713 ns	724 ns
tcmalloc	42% / 22475.24%	2.50%	18.12%	1840 ns	799 ns
dlmalloc	51% / 39241.38%	5.00%	5.62%	1007 ns	897 ns
rmmalloc-c-m	68% / 56677.64%	0.00%	60.00%	982 ns	1022 ns

Table 8.5: Speed measurements for ls

	Space		
Driver	**Penalty (c/w)**	**Best**	**Worst**
tcmalloc	0% / 0.00%	100.00%	0.00%
rmmalloc-c-m	24% / 5.83%	0.00%	0.00%
rmmalloc	41% / 9.35%	0.00%	0.00%
dlmalloc	42% / 6.00%	0.00%	0.00%
rmmalloc-c	58% / 12.74%	0.00%	0.00%
jemalloc	83% / 82.75%	0.00%	100.00%

Table 8.6: Space measurements for ls

Starting with the speed table we see similarly to previous measurements where *jemalloc* failed early, that the absolute results are skewed but the internal order is still correct. Good performance of *rmmalloc* but also of *tcmalloc* which has differing results. As for memory efficiency, *tcmalloc* stands out after which the results for *rmmalloc* and *dlmalloc* are very similar. *rmmalloc-c-m* fares slightly better, but is on the other hand very time consuming. This might not be a trade-off the client code can make.

8.4.4 tar with bzip2 compression

Command line used: `tar cjf /tmp/valgrind-3.9.0.tar.bz2 /tmp/valgrind-3.9.0`

Simulated using full lockops.

Results in Figure 8.4, Table 8.7 and Table 8.8.

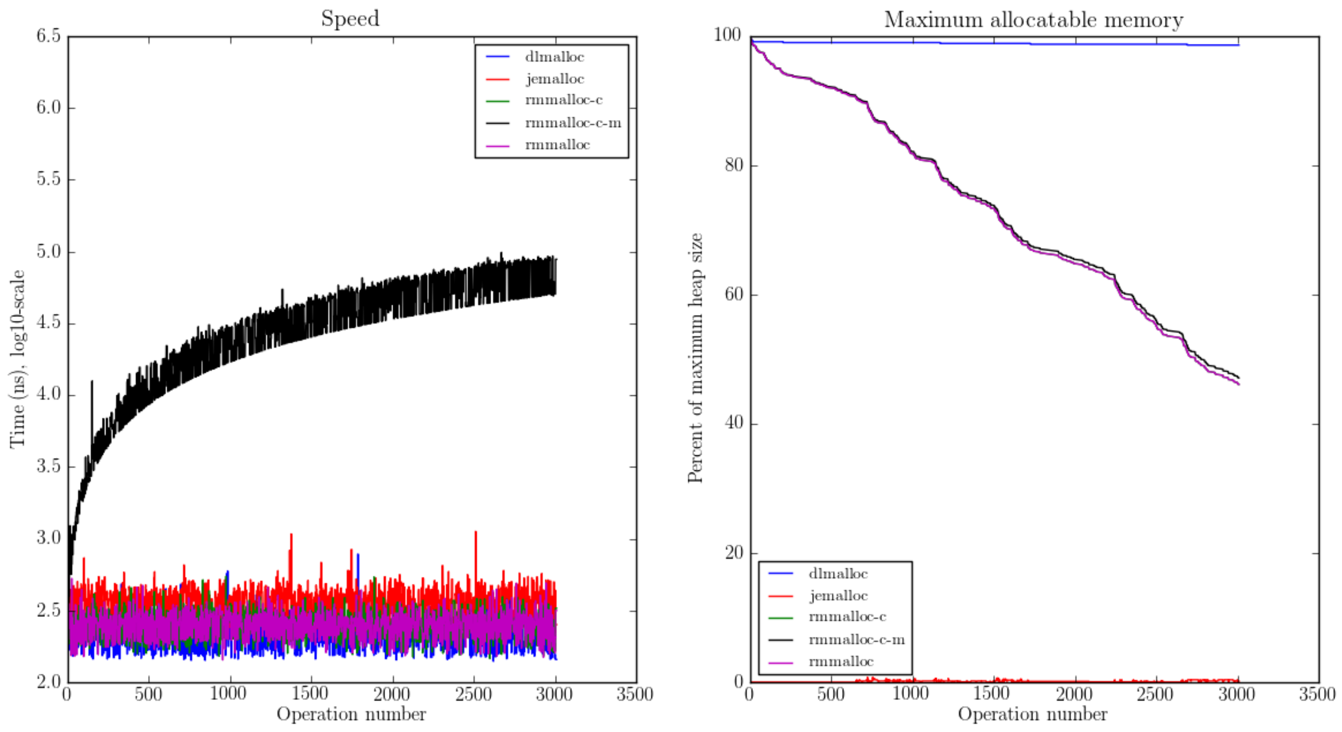


Figure 8.4: tar cjf results.

For the linearly growing allocation pattern used in tar, *rmalloc-c-m* does not fare well with its exponential algorithm. The others are segmented, with *dlmalloc* coming out as the fastest, followed by *rmmalloc*. As for memory efficiency, *dlmalloc* is the clear winner here.

Speed					
Driver	**Penalty (c/w)**	**Best**	**Worst**	**Average**	**Median**
dlmalloc	15% / 5.73%	50.96%	0.00%	233 ns	235 ns
rmmalloc-c	26% / 12.71%	23.06%	0.00%	257 ns	258 ns
rmmalloc	26% / 12.06%	23.19%	0.00%	256 ns	256 ns
jemalloc	50% / 100.08%	2.79%	0.37%	1228 ns	365 ns
rmmalloc-c-m	79% / 15087.61%	0.00%	99.63%	36592 ns	34975 ns

Table 8.7: Speed measurements for tar

Space			
Driver	**Penalty (c/w)**	**Best**	**Worst**
dlmalloc	0% / 0.00%	99.93%	0.00%
rmmalloc-c-m	19% / 5.15%	0.07%	0.00%
rmmalloc	39% / 10.49%	0.00%	0.00%
rmmalloc-c	59% / 15.74%	0.00%	0.00%
jemalloc	80% / 79.89%	0.00%	100.00%

Table 8.8: Space measurements for tar

There are no real surprises in speed in Table 8.7, since the graphs are easy to interpret directly. Here it's important to note that even though the space numbers in Table 8.8 look good enough for the *rmmalloc* allocator (and variants), it's still performs a lot worse than *dlmalloc*. It is not sufficient to look only at the numbers.

8.4.5 latex

Command line used: `latex paper.tex`

Simulated using full lockops.

Results in Figure 8.5, Table 8.9 and Table 8.10 (tcmalloc did not finish).

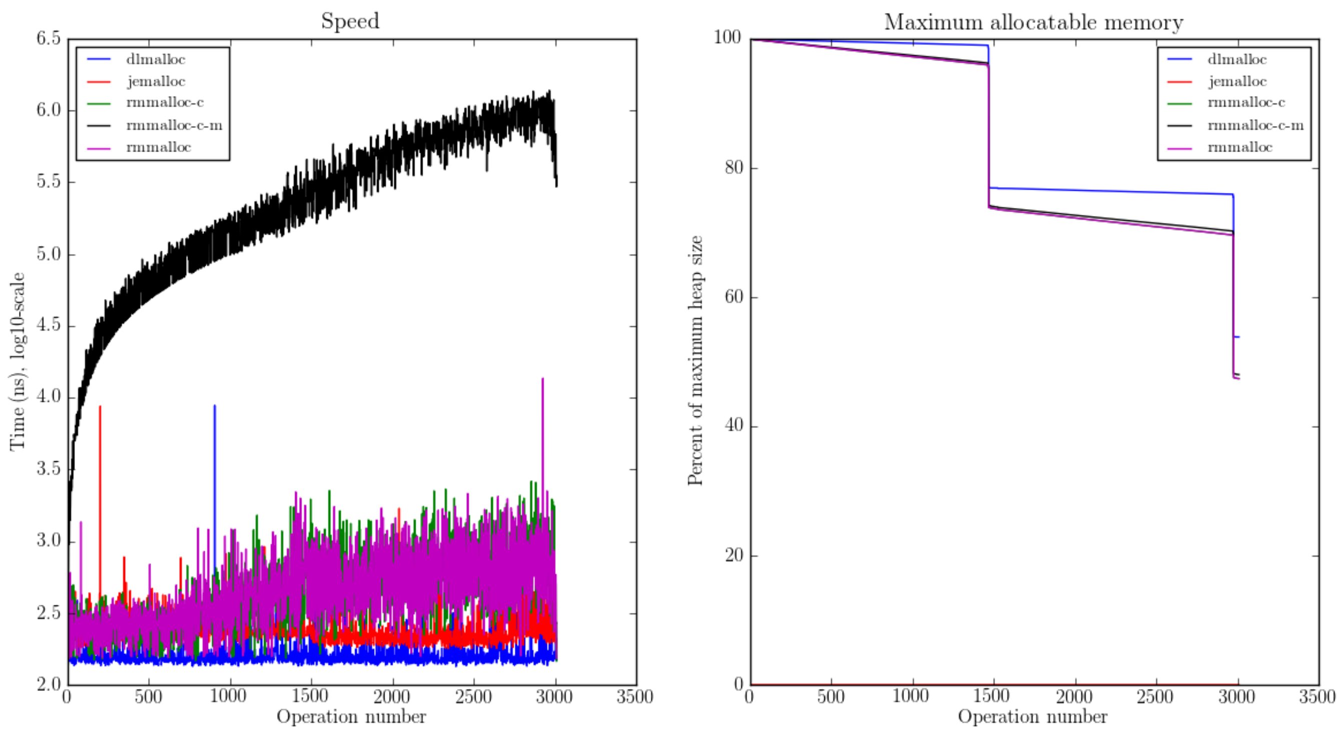


Figure 8.5: latex results.

A very simple linear allocation pattern, where we clearly see the time inefficiency of *rmalloc-c-m* because of its exponential search. It does however fare well when it comes to space. Even *rmalloc-c* which has fairly good performance in time still has a exponential tendency whereas *dlmalloc* is mostly linear. Here's a case where the trade-off might not be worth it, especially since *dlmalloc* performs better in both areas.

Speed					
Driver	**Penalty (c/w)**	**Best**	**Worst**	**Average**	**Median**
dlmalloc	1% / 1.54%	93.49%	0.00%	167 ns	152 ns
jemalloc	23% / 65.43%	4.65%	0.03%	621 ns	224 ns
rmmalloc-c	46% / 122.33%	1.13%	0.00%	523 ns	428 ns
rmmalloc	47% / 124.81%	0.73%	0.00%	530 ns	417 ns
rmmalloc-c-m	79% / 189701.98%	0.00%	99.97%	372546 ns	268695 ns

Table 8.9: Speed measurements for latex

Space			
Driver	**Penalty (c/w)**	**Best**	**Worst**
dlmalloc	0% / 0.00%	99.93%	0.00%
rmmalloc-c-m	19% / 0.72%	0.03%	0.00%
rmmalloc	39% / 1.60%	0.00%	0.00%
rmmalloc-c	59% / 2.40%	0.03%	0.00%
jemalloc	80% / 80.00%	0.00%	100.00%

Table 8.10: Space measurements for latex

No surprises here since the graphs are easy to read for this test case.

8.4.6 opera

Command line used: `opera`

Due to memory/CPU constraints, I was not able to perform a locking data calculation. The results are therefore without any locking/unlocking, which means that any compacting operations are optimal (no locked blocks).

Results in Figure 8.6, Table 8.11 and Table 8.12.

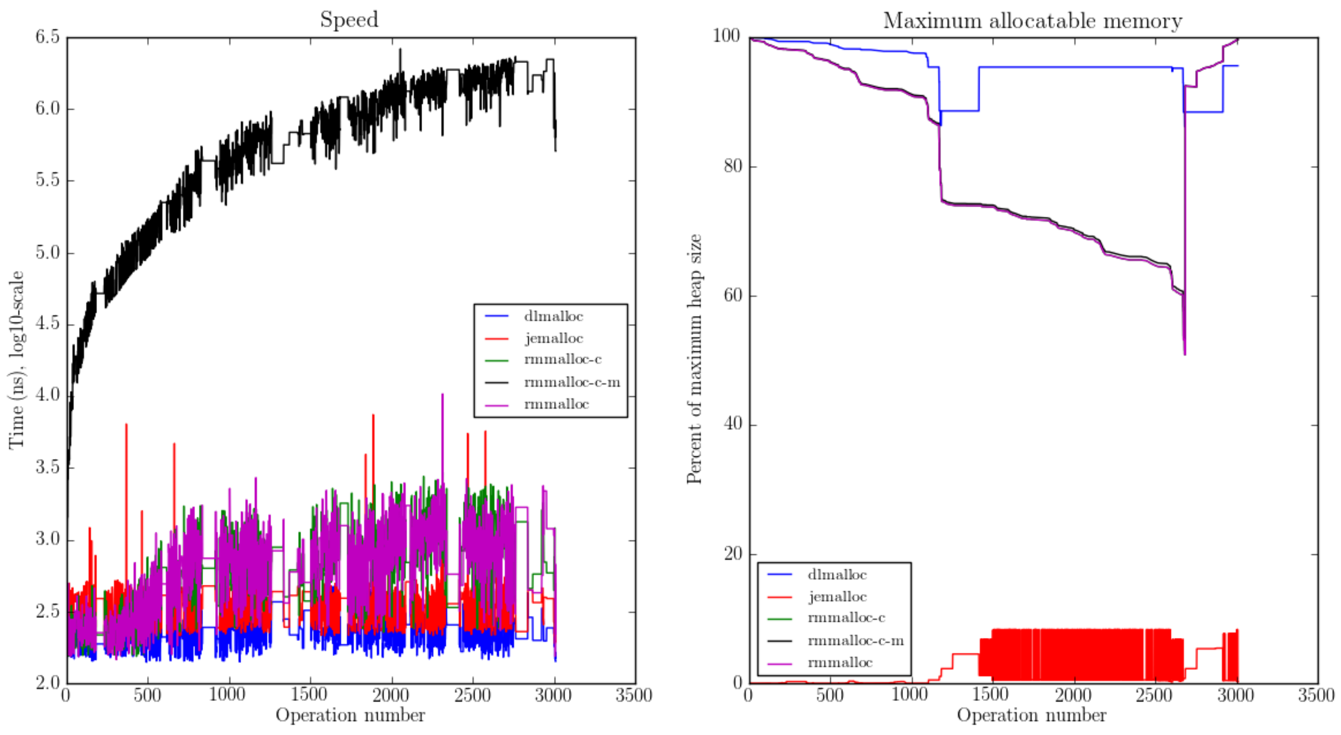


Figure 8.6: Opera results.

We see the same characteristics as in the LaTeX test above, except for the ranges where no allocation operations happen. *jemalloc* performs well speed-wise but badly in available space.

	Speed				
Driver	**Penalty (c/w)**	**Best**	**Worst**	**Average**	**Median**
dlmalloc	4% / 1.75%	83.19%	0.00%	239 ns	233 ns
jemalloc	26% / 125.39%	9.37%	0.03%	1000 ns	372 ns
rmmalloc-c	44% / 124.86%	2.82%	0.00%	752 ns	632 ns
rmmalloc	44% / 132.46%	4.62%	0.00%	779 ns	704 ns
rmmalloc-c-m	79% / 310598.14%	0.00%	99.97%	861988 ns	687897 ns

Table 8.11: Speed measurements for opera

	Space		
Driver	**Penalty (c/w)**	**Best**	**Worst**
dlmalloc	6% / 0.36%	89.20%	0.00%
rmmalloc-c-m	19% / 2.88%	0.03%	0.00%
rmmalloc	35% / 5.88%	0.00%	0.00%
rmmalloc-c	57% / 8.82%	10.76%	0.00%
jemalloc	80% / 78.51%	0.00%	100.00%

Table 8.12: Space measurements for opera

Again, skewed results because of *jemalloc*. By far fastest and most space-efficient is *dlmalloc* in this type of scenario.

Chapter 9

Conclusions

9.1 Speed

Calculate the penalty for the fields *penalty*, *best*, *worst* and *average* per application, which gives each allocator a sum of penalties for each field. By taking the average of these penalties, we can tell the position of each allocator. Allocators that did not finish are given the maximum penalty 5.

This is summarized in Table 9.1 below, and we can make a final scoring of the allocators:

1. rmalloc-c (Jeff: compacting)
2. rmalloc (Jeff: plain)
3. dlmalloc
4. jemalloc
5. tcmalloc
6. rmalloc-c-m (Jeff: compacting, maxmem)

			Speed		
Driver	**Penalty**	**Best**	**Worst**	**Average**	**Average penalty**
rmalloc	12	14	2	7	8.8
rmalloc-c	8	9	2	5	6.0
rmalloc-c-m	27	27	20	22	24.0
dlmalloc	12	11	6	8	9.3
jemalloc	8	9	11	23	12.8
tcmalloc	23	22	24	23	23.0

Table 9.1: Positions of allocators for speed

9.2 Memory

Calculated the same way as speed. Because of the extra indirection layer, there will always be more memory used per allocated block. Summary in Table 9.2 below with scoring of the allocators:

1. dlmalloc
2. rmalloc-c-m (Jeff: compacting, maxmem)
3. rmalloc (Jeff: plain)
4. rmalloc-c (Jeff: compacting)
5. tcmalloc
6. jemalloc

Memory				
Driver	**Penalty**	**Best**	**Worst**	**Average penalty**
rmalloc	13	10	0	7.6
rmalloc-c	21	7	0	9.3
rmalloc-c-m	7	7	0	4.6
dlmalloc	7	3	0	3.3
jemalloc	27	9	6	14.0
tcmalloc	15	13	10	12.6

Table 9.2: Positions of allocators for memory

9.3 Discussion

Important to note when making a decision on which allocator to use.is that tcmalloc was not able to finish all tests. Most of the tested allocators were designed to use `mmap()` for memory allocation along with `sbrk()` which likely skewed the results. In particular jemalloc performs badly, which could be caused by it being optimized for `mmap()`.

Noteworthy is that dlmalloc still performs better than Jeff with compacting and specific support for maximum available memory. It is possible that fitting Jeff's interface on top of an existing tested and quick allocator, e.g. dlmalloc, would have given better runtime characteristics in both space and time. Jeff is a very simplistic implementation of a buddy-style allocator without any pools for small objects and similar techniques found in most modern allocators.

Another conclusion to be drawn from the graphs is that there are cases where a fairly naive allocator, such as Jeff, still performs almost as well as a more complex allocator, such as dlmalloc. There might be cases where the trade-off in code size versus memory efficiency and speed might be worth it, e.g. when the amount of code storage media is limited, again, common in embedded systems with only kilobytes of code ROM.

Jeff still does perform quite well, which means the idea itself could be expanded on in the future. Due to time constraints, larger applications that are more similar to real-life situations could not be tested since the lockops calculation took too long time. Speed and memory characteristics could very well differ for such an application, especially if it were to run for a longer time.

9.4 Limitations and Future Work

9.4.1 Jeff: Limitations

In order to keep the code simple, I made two decisions in the beginning:

- The allocator does not align memory of allocated chunks to boundaries. On older computer architectures, accessing non-aligned memory will cause an access violation. In newer architectures, the code runs with a small speed penalty.
- No thread-safety. This means that the behaviour of calling any functions exposed by the allocator from different threads at the same time is undefined, and will likely cause data corruption.

9.4.2 Jeff: Future Work

Features

- Add a callback when moving a locked block, for simpler compact operation and client code so that memory does not have to be locked/unlocked manually. Instead, they could be locked during their entire lifetime. On the other hand, there is a risk that it would lead to the lookup table being on the client side instead of in the allocator. Depends on use case.
- Use bits of pointer to memory block, if size is limited. In practice, a special-purpose allocator such as Jeff will likely work with less than the full 32 bits. (For example, limiting to max 1 GB heap gives two extra bits for flags.)
- Weak locking
- Introduce a mature generation for blocks that have been locked for n compactions. This would require application co-operation in updating any references to the block.

Implementation Optimizations

- Similar to the earlier point, reduce `next_unused` to store offset into a heap array. This limits the maximum number of live blocks to $2^{sizeof(next\_unused\_offset)}$, which might not be an issue. It could be a compile-time setting.
- Automatic merge with adjacent previous/next block in `free`/`new`. This would however cause the free list slots contain too large blocks for its index.
- Quick free block find overwrites itself, issue #1 in the Github rmalloc issue tracker.

9.4.3 Steve: Limitations

As noted in the discussion, the only mechanism for retrieving data from the system for the tested allocators is using `sbrk()`. Moreover, there are no hand-tuned reference applications, where optimal locking/unlocking is manually inserted. This was not done because of time constraints, but would be interesting to do in the future to establish a baseline to which other allocators could be compared.

9.4.4 Steve: Future Work

Simplification

- Simplify running tests, specifically setting `CORES`, `ALLOCATOR` and `KILLPERCENT`.
- Load allocators as shared libraries instead of linking to `plot.cpp`.
- Restart simulation
- Don't use part files, if possible.

Features

- Reintroduce colormap for calculating theoretical free size from overhead marked in the colormap.
- Measure how large part of the total number of blocks are locked at compacting time.
- Investigate stack-based behaviour of computation (and thus allocation) for a possibly more realistic heuristic for calculating locking.

Chapter 10

References

- *Garbage Collection - Algorithms for Automatic Dynamic Memory Management* (Richard Jones, Rafael Lins, 1997, WILEY PRESS)
- *Analysis on Dynamic Memory Allocation* (Wenbin Fang, May 2012)
- *The Memory Fragmentation Problem: Solved?* (Mark S. Johnstone, Paul R. Wilson, 1998)
- *A Scalable Concurrent malloc(3) Implementation for FreeBSD* (Jason Evans, April 2006) - `http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf`
- *Memory Management in the Java HotSpot Virtual Machine* (Sun Microsystems, April 2006) - `http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf`

Chapter 11

Appendix

The report and the source code in its entirety can be found on GitHub, `http://github.com/mikaelj/rmalloc`

11.1 Tools

11.1.1 memtrace-run.sh and translate-memtrace-to-ops.py

Generates memtrace data from an application run, and translates memtrace data to ops file, respectively, as described in section 5.2.

11.1.2 translate-ops-to-histogram.py

To visualize and experiment with different ways of calculating lifetime I have a small application that takes as input an ops file (created by `translate-memtrace-to-ops.py`), to visualize macro lifetime in different intervals. This is described in Section 5.3.

11.1.3 translate-ops-to-locking-lifetime.py

`translate-memtrace-to-ops.py` produces coarse locking that is quick to calculate, since it simply looks at the macro lifetime of an object and keeps it locked during its entire lifetime. This is done instead of locking and unlocking throughout the object lifetime.

11.1.4 run_memory_frag_animation.sh

Syntax:

```
ALLOCATOR=path/to/alloc_driver \
    ./run_memory_frag_animation.sh opsfile
```

Example:

```
ALLOCATOR=./drivers/plot_dlmalloc \
    ./run_memory_frag_animation.sh result.soffice-ops
```

Output:

```
result.soffice-ops-animation.avi
```

The tool calls the *memplot* mode described above and calls *ffmpeg* to generate an animation of the heap image sequence produced by the alloc driver for the given ops file.

11.1.5 run_allocator_stats.sh

Run:

```
CORES=2 ALLOCATOR=./drivers/plot_dlmalloc \
    ./run_allocator_stats.sh result.soffice-ops
```

Generates:

```
result.soffice-ops.allocstats
```

11.1.6 run_graphs_from_allocstats.py

From data created by run_allocator_stats.

Single

Run:

```
python run_graphs_from_allocstats.py result.soffice-ops
```

Generates:

```
plot-<driver>-<opsfile>.png
```

Multiple

Run:

```
python run_graphs_from_allocstats.py soffice \
    result.soffice-ops-dlmalloc \
    result.soffice-ops-rmmalloc [...]
```

Generates:

soffice.png

11.2 Allocator driver API

This gives the essentials of a program's memory usage -- allocation, access and free -- and can be processed by other tools.

Testing an allocator is done with a driver application by implementing an interface that calls the appropriate functions of the allocator and linking to a library. The functions to implement are:

```
bool user_init(uint32_t heap_size,
               void *heap,
               char *name);
void user_destroy();
bool user_handle_oom(int size,
                     uint32_t *op_time);
void *user_malloc(int size,
                  uint32_t handle_id,
                  uint32_t *op_time,
                  void **memaddress);
void user_free(void *handle,
               uint32_t handle_id,
               uint32_t *op_time);
void *user_lock(void *handle);
void user_unlock(void *handle);
void *user_highest_address(bool full_calculation);
```

All functions to be implemented by the driver have a `user_` prefix and the driver code is linked together with `plot.cpp` to form the binary.

11.2.1 user_init(heap_size, heap, name)

`bool user_init(uint32_t heap_size, void *heap, char *name)`

Initialize the allocator with the given parameters. Since the heap is passed onto the driver, any *mmap* functionality must be disabled and only *sbrk*-style allocation is possible. The driver must set `name` to a string that can be used as a part of a filename, e.g. an alphanumeric string like "dlmalloc".

A driver initializes its own sbrk-equivalent with *heap* and *heap_size* and initializes the allocator itself if needed. As large amount as possible of the allocator's runtime data structures should be stored in this heap space.

11.2.2 user_destroy()

`void user_destroy()`

Clean up internal structures. The heap given to `user_init` is owned by the framework and does not have to be freed.

11.2.3 user_handle_oom(size, op_time)

`bool user_handle_oom(int size, uint32_t *op_time)`

Handle an out-of-memory situation. `size` is the number of bytes requested at the time of OOM. `op_time` is an out variable storing the time of the actual OOM-handling code (such as a compact operation), not considering the code before or after. For convenience, Steve defines macros for time measuring. A typical implementation where OOM is actually handled looks like this:

```
bool user_handle_oom(int size, uint32_t *op_time)
{
    TIMER_DECL;

    TIMER_START;
```

```
        bool ok = full_compact();
        TIMER_END;
        if (op_time)
            *op_time = TIMER_ELAPSED;

        return ok;
    }
```

`op_time` can also be `NULL`, as shown in the example, in which case time must not be stored. Return value is *true* if the OOM was handled, *false* otherwise.

11.2.4 user_malloc(size, handle, op_time, memaddress)

`void *user_malloc(int size, uint32_t handle, uint32_t *op_time, void **memaddress)`

Perform a memory allocation and return a pointer to the allocated memory, or `NULL` on error. `op_time` is the same as above. `handle` is an identifier for this allocation request as translated from the memtrace, unique for this block for the lifetime of the application being benchmarked. It can be used as an index to a map in case the driver wants to store information associated with this particular block. Finally, `*memaddress` can be used to store the memory address at the time of the allocation, in case the allocation function is using indirect accessing via a handle (e.g. Jeff). In that case, the handle is returned by *user_malloc()* and the memory address stored in `*memaddress`. If *memaddress* is `NULL` no data should be written to it, but if it is not `NULL`, either the address or `NULL` should be stored in `*memaddress`.

11.2.5 user_free(ptr, handle, op_time)

`void user_free(void *, uint32_t handle, uint32_t *op_time)`

Like `user_malloc`.

11.2.6 user_lock(ptr)

`void *user_lock(void *)`

This locks a block of memory: map a handle to a pointer in memory, and mark the block as in use. It can no longer be moved since the client code now has a reference to the memory referred to by this handle, until `user_unlock()` or `user_free()` is called on the handle. Its input value is the return value of `user_malloc()`.

11.2.7 user_unlock(ptr)

`void user_unlock(void *)`

This unlocks a block of memory, i.e. marking the block of memory as no longer being in use. Any memory operation is free to move this block around in memory. Its input value is the return value of `user_malloc()`.

11.2.8 user_highest_address(fullcalc)

```
void *user_highest_address(bool full_calculation)
```

What is the highest address allocated at this time? `NULL` if not available. If `full_calculation` is false a less exact calculation is acceptable if it's quicker.