OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

Advanced Topics in Databases
Database and Software Engineering Working Group

Magdeburg, May 2nd 2019

Summer Term 2019

Marcus Pinnecke, M.Sc.
Prof. Dr. Gunter Saake

Optimizing Memory Pooling for In-Memory Database Systems

submission deadline: **June 2nd 2019** 11:59pm

In this programming exercise you will get your hands dirty with a particular component of our main-memory database system, the *memory pool*. You will exercise several task related to low-level database system research and development.

This is a per-group sheet with no restriction on the number of groups that choose this task. Your complete submission (containing of *your code*, final plots "*Memory Pool Overhead*", "*Memory Pool Performance Improvement*", and "*Magic Memory Pool Performance*") must be online accessible for your tutor until the submission deadline (e.g., in a repository like GitHub.com).

Don't forget to share access to your submission with, and send mails with your deliveries to pinnecke{at}ovgu.de right in time!

You must have 3.5 of 7 points to pass.

Good Luck!

Programming Exercise Sheet

After completing this sheet, you will

- ✓ get more quickly familiar with a larger code base of a non-trivial system
- ✓ have extended knowledge about memory management & system-level development in C (C11 standard)
- ✓ have a better understanding of practical research and processes
- ✓ be more sensible to performance related optimizations considering multiple dimensions (runtime, memory consumption,...)
- ✓ be more trained to find solution concepts by yourself, and to implement these concepts as a productive solution
- ✓ have improved your skills related to evaluating, discussing, judging, and visualizing solutions quantitatively

Context. In our inhouse analytic document storage engine (called *NG5*), we have the need for non-standard dynamic memory management to satisfy real-time processing requirements for parsing and modifying huge amounts of JSON plain-text data, object management, and others. For these cases, we use a specialized memory manager, the *memory pool*. Our memory pool is an extensible memory allocator built from scratch to support *bulk memory releases*, and *automatic memory management* (garbage collection). Additionally, fast (re-)allocations, and memory releases are supported by exploiting x86-64 architecture features and modern hardware capabilities.

Task Overview. Imagine you are collaborating to the memory pool in its early stages of development. So far, the extendible architecture of the memory pool is designed and implemented, and (re-)allocations, (bulk) memory releases work just fine with a straight-forward delegation to the standard memory allocation of the C library (garbage collection is plan for the next milestone). Now, you are investigating the memory pools overhead for mixed realloc/free workloads on a fixed number of pre-allocated memory blocks by comparing to the standard memory allocator. As it turns out, despite the benefit of managed memory management with the memory pool, an undesired overhead makes the memory pool uncompetitive to its alternative. Your task is to *find*, and *implement* a solution to this issues. To support your claim of solving the issue, you show a *performance plot* of your solution comparing to the standard allocator. Finally, you highlight *key ideas*, *state assumptions*, and *discuss benefits and drawbacks*.

Task 1 Getting the Sources, Repeating & Preparation 1.5 Points

For this exercise you will have to do a bit of setup first.

1. **Register.** Start with sending an e-mail to pinnecke{at}ovgu.de with subject *ATDB-MemPool REQUEST* containing your group mate names, e-mail-addresses and matriculation numbers. You will get a mail in return containing a unique group name token *XXX.* You must use this token during the course of this exercise sheet to identify your group (0.5 Points)

   Tip: Start early

2. **Setup.** Download, compile and run the benchmark system (see "Getting Started" guide below). During this, three plots are created. Send the plot "Memory Pool Overhead" to pinnecke{at}ovgu.de with the subject *ATDB-MemPool XXX Task 1.2.* In this e-mail, provide information about the machine on which you performed the benchmark, i.e., state the operating system (vendor, name and kernel version) and hardware specs (CPU, RAM capacity, DRAM type,...) (1 Point)

   Tip: Don't underestimate download, setup and orientation time. Plan at least 30min.

The following is a "Getting Started" guide to lead you through the setup. You need to get the sources of NG5, compile the system, start a already prepared benchmark, and plot the results. We recommend to use Ubuntu Linux or macOS as operating system.

Getting the sources and compile the system

On a fresh installed Ubuntu 18.04.2 LTS, run the following commands in your bash

```
$ sudo apt install -y git  cmake clang r-base
$ git clone https://github.com/protolabs/libcarbon.git
$ cd libcarbon/
$ git checkout -b teaching/atdb/2019 origin/teaching/atdb/2019
$ cmake . && make -j 8 && make benches
$ mkdir benches/mem/pools/results
```

After these commands have been completed, you should find an executable called `bench-mem-pools` in the directory `build/`. This executable is the benchmark runner that supports you on your way.

When running the benchmark tool without parameters, it echos its usage instructions

```
$ build/bench-mem-pools
usage: <allocator>

<allocator> is the identifier of an allocator implementation to bench.
Use 'clib/allocator' to benchmark clibs allocator, and <mem pool names>
to bench those implementations.

The following <mem pool names> are registered:
        'mempool/none'
        'mempool/magic'
```

With the argument `clib/allocator`, the benchmark runs the experiment with the standard Clib allocator (`malloc`, `realloc`, and `free`). When running with argument `mempool/none`, the benchmark runs a trivial memory pool strategy implementation that just delegates calls to the underlying standard CLib allocator, and that performs the required communication with the memory pool itself. This memory pool strategy should show you the minimum example on an implementation of a strategy; it is documented exhaustingly to help you on your way.

Repeating (showing) the performances of the memory pool, and your solution

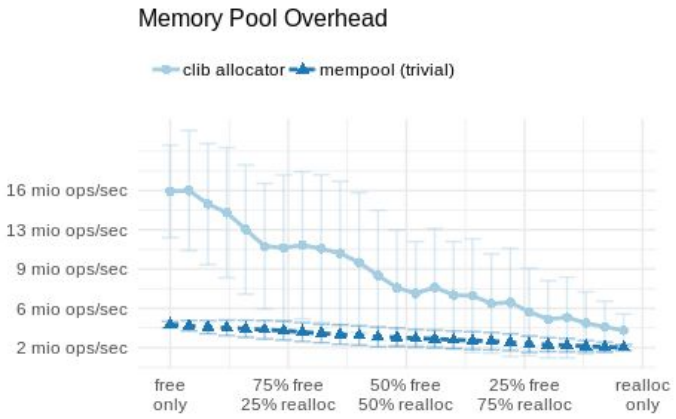
When running the benchmark tool with `clib/allocator` and `mempool/none`, you are able to investigate the overhead that comes from the memory pool architecture (e.g., the impact of indirection layers). To mimic your investigation (that would be done by you upfront), you must run the benchmark tool, and store its results as CSV files. You feed prepared) R scripts with these CSV files. After running these R scripts, you should see a plots similar to those below.

The entire process of performance overhead measurement is prepared for you. Run the following in your bash after you compiled the system properly to generate benchmark results:

```
$ build/bench-mem-pools clib/allocator >
  benches/mem/pools/results/results-bench-mem-pools-clib-allocator.csv
$ build/bench-mem-pools mempool/none >
  benches/mem/pools/results/results-bench-mem-pools-mempool-none.csv
```

Each call to `build/bench-mem-pools` will take a few minutes to complete (around 7min).

Memory Pool Overhead Plot. Afterwards, run your favored R environment (e.g., R Studio[1]), install required packages if needed, and open the R script `show-overhead-mempool-vs-clib.r` stored in `benches/mem/pools/r-scripts/`. This script reads `results-bench-mem-pools-clib-allocator.csv` and `results-bench-mem-pools-mempool-none.csv` stored in `benches/mem/pools/results`, and produce the following plot:



This plot is *your investigation of the memory pools overhead for mixed realloc/free workloads on a fixed number of pre-allocated memory blocks by comparing to the standard memory allocator*. As expected , memory reallocation is more expensive than memory releasing. This leads to a higher performance for "free only" workloads and a lower performance for "realloc-only" workload. In the mixed case, you also see the increasing impact of reallocation cost. You also note that there is *no* case in which the memory pool is actually outperforming the standard allocator.

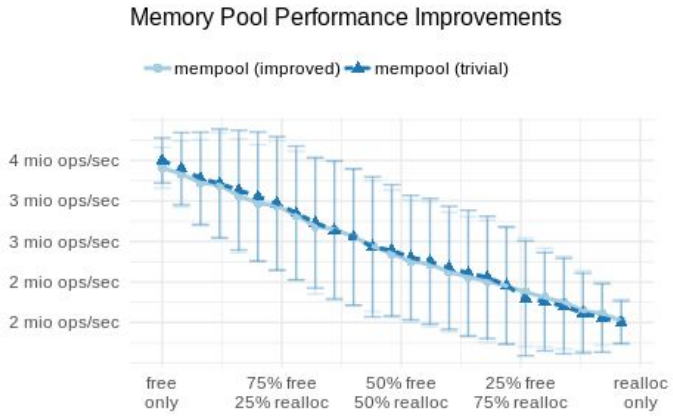
Memory Pool Performance Improvements Plot. Before you start, you want to make sure that you always see the effect of your changes. For this purpose, there is a prepared memory pool strategy (`mempool/magic`) that you will work with. To benchmark the performance of your `mempool/magic` memory pool strategy at any time, run

```
$ build/bench-mem-pools mempool/magic >
  benches/mem/pools/results/results-bench-mem-pools-mempool-magic.csv
```

Run the r-script `compare-magic-vs-trivial.r` stored in `benches/mem/pools/r-scripts/` that compares the benchmark results for `mempool/magic (results-bench-mem-pools-mempool-magic.csv)` with the results for the trivial implementation `mempool/none (results-bench-mem-pools-mempool-none.csv)`.

[1] https://www.rstudio.com/products/rstudio/download/#download

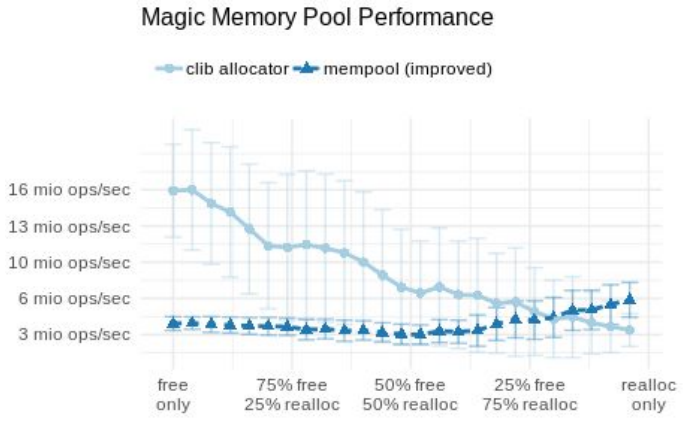
Initially, you will see a plot similar to this after running the script:



This plots shows the performance of the trivial pool strategy (`mempool/none`) compared to the current state of the pool strategy (`mempool/magic`) that you are working on, labeled "mempool (improved) in the plot. During the course of this sheet, you will modify the logic of `mempool/magic` which leads to changes in this plot. Your final submission must include the latest plot showing your final changes to the memory pool performance.

Magic Memory Pool Performance. Your goal is to improve the performance of the memory pool by improving the `mempool/magic` pool strategy. To show this improvement compared to the standard Clib allocator, run the R-script `magic-memory-pool-performance.r` stored in `benches/mem/pools/r-scripts/` that compares the benchmark results for `mempool/magic` (`results-bench-mem-pools-mempool-magic.csv`) with the results of the standard Clib allocator `mempool/none` (`results-bench-mem-pools-clib-allocator.csv`).

Depending on your actual solution and implementation in `mempool/magic`, your final plot may look like this:



Your final submission must include the latest plot showing your final comparison to the standard allocator.

Preparation for Development

To start improving the `mempool/magic` pool strategy, you need to get familiar with our memory pool, its architecture and the trivial implementation which performance you just plotted. For this, it is recommended to install a proper C integrated development environment (if needed). We recommend *CLion* from JetBrains[2], since it smoothly integrates with the tool chain of this project. Open the project with the `CMakeLists.txt` file located in the repository root. Add a run configuration with target `bench-mem-pools` and set as program arguments either `clib/allocator`, `mempool/none`, or `mempool/magic` (depending on which strategy you want to run), and run this target .

> **Tip:** If you want to understand the entire memory pool (rather than a single strategy only), set as program argument `mempool/none` (or `mempool/magic`), add a breakpoint in the `main` function of `benches/mem/pools/main.c`, and perform stepwise debug run.

[2] https://www.jetbrains.com/clion/download/

Task 2 Improving the Memory Pool 3.5 Points

For this exercise, you have to improve the performance of the memory pool implementation. Despite the fact that the memory pool architecture introduces a particular overhead to enable bulk-releases of memory blocks and garbage collection, minimizing the number of system calls for allocating (via `malloc`), reallocating (via `realloc`) and releasing of memory blocks (via `free`) are the major tuning option to increase the memory pool performance.

You will improve the performance by executing the following tasks:

1. **Understand the Pool Strategy**. In the appendix of this sheet, you find a print of an excerpt of the implementation code of the trivial pool strategy `mempool/none`. Describe in your own words what happens in the functions `this_alloc`, `this_realloc`, and `this_free`. Send your description as via mail to pinnecke{at}ovgu.de with the subject *ATDB-MemPool XXX Task 2.1* right in time. (1.5 Points)

   Tip: refresh your knowledge on C (if needed) with this tutorial:
   youtube.com/playlist?list=PLGLfVvz_LVvSaXCpKS395wbCcmsmgRea7
   Tip: see www.cplusplus.com/reference/cstdlib/malloc/ for Clib `malloc`
   Tip: see www.cplusplus.com/reference/cstdlib/realloc/ for Clib `realloc`
   Tip: see www.cplusplus.com/reference/cstdlib/free/ for Clib `free`
   Tip: Have a look at `src/include/core/ptrs/data_ptr.h`
   Tip: Have a look at the actual implementation file `src/core/mem/pools/none.c`

2. **Concept and Evaluation.** Your task is to provide and implement a concept of *your* choice to improve the memory pool performance at least for one particular workload mix, e.g., for reallocation-heavy workloads, cf. *Magic Memory Pool Performance Plot* from Task 1. For this you must modify the logic of the "Magic Pool Strategy" (`src/core/mem/pools/magic.c`).

   To apply a modification, you must recompile the target `bench-mem-pools` by running

   ```
   $ cmake . && make -j 8 && make benches
   ```

   again. Finally, you must re-evaluate your benchmark results by running

   ```
   $ build/bench-mem-pools mempool/magic >
     benches/mem/pools/results/results-bench-mem-pools-mempool-magic.csv
   ```

   again, and plot the *Memory Pool Performance Improvements*, and the *Magic Memory Pool Performance* with R again (as describe in Task 1 "Get Started" guide).

   Share your code via a code repository (e.g., GitHub) with pinnecke{at}ovgu.de with subject *ATDB-MemPool XXX Task 2.2* right in time. In your repository, also store both plots the *Memory Pool Performance Improvements* and *Magic Memory Pool Performance* plot, and give a description in your mail where to find these plots in your repository (2 Points).

   Tip: The memory pool internally book-keeps information about managed pointers independent of which pool strategy is used. In particular, the following information are available: (1) the allocated memory block size (`bytes_total`), and (2) the block size used by the caller (`bytes_used`), where `bytes_used <= bytes_total` holds. Implement the case in which *no* `realloc` call is required.

   Tip: There exists a "quick and dirty" solution with 3 lines of code.

Task 3 Explaining, Discussing and Judging 2 Points

In this final task, you must explain, discuss and judge your solution.

3. **Reasoning.** Send an e-mail to pinnecke{at}ovgu.de with subject *ATDB-MemPool XXX Task 3* right in time. In this mail, you state what you did to improve the performance, why you did it, what your assumption where, and what benefits and drawbacks result from your solution (e.g., with focus of memory consumption) (2 Points)

Appendix Implementation Trivial Pool Strategy (mempool/none)

src/core/mem/pools/none.c

```
#include "core/mem/pool.h"

#define REQUIRE_INSTANCE_OF_THIS() ng5_check_tag(self->tag, POOL_IMPL_NONE);

/* ... */

static data_ptr_t this_alloc(struct pool_strategy *self, u64 nbytes)
{
        REQUIRE_INSTANCE_OF_THIS()

        void *ptr = malloc(nbytes);
        assert(ptr);

        self->counters.num_alloc_calls++;
        self->counters.num_bytes_allocd += nbytes;

        return pool_internal_new(self, ptr, nbytes);
}

static data_ptr_t this_realloc(struct pool_strategy *self, data_ptr_t ptr, u64 nbytes)
{
        REQUIRE_INSTANCE_OF_THIS()

        void *stored_adr, *new_adr;
        struct pool_ptr_info *info;

        info = pool_internal_get_info(self, ptr);

        self->counters.num_bytes_reallocd = nbytes;
        self->counters.num_bytes_allocd += ng5_span(info->bytes_total, nbytes);

        stored_adr = data_ptr_get_pointer(ptr);
        new_adr = realloc(stored_adr, nbytes);

        if (unlikely(!new_adr)) {
                error_print(NG5_ERR_REALLOCERR);
        } else {
                data_ptr_update(&ptr, new_adr);
                data_ptr_update(&info->ptr, new_adr);
                info->bytes_used = nbytes;
                info->bytes_total = nbytes;
                self->counters.num_realloc_calls++;
        }

        return ptr;
}

static bool this_free(struct pool_strategy *self, data_ptr_t ptr)
{
        REQUIRE_INSTANCE_OF_THIS()

        void *adr;
        struct pool_ptr_info *info;

        info = pool_internal_get_info(self, ptr);
        adr  = data_ptr_get_pointer(ptr);

        free (adr);
        pool_internal_delete(self, ptr);

        self->counters.num_free_calls++;
        self->counters.num_bytes_freed += info->bytes_total;

        return true;
}
```