**FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE**

# ASSIGNMENT OF MASTER’S THESIS

| | |
|---|---|
| **Title:** | Performance analysis of the LSU3shell program |
| **Student:** | Bc. Martin Kočička |
| **Supervisor:** | Ing. Daniel Langr, Ph.D. |
| **Study Programme:** | Informatics |
| **Study Branch:** | System Programming |
| **Department:** | Department of Theoretical Computer Science |
| **Validity:** | Until the end of summer semester 2018/19 |

## Instructions

1) Get familiar with the problem and existing solutions of dynamic memory allocations (glibc malloc, jemalloc, tcmalloc, tbbmalloc, or others).
2) Get familiar with existing implementations of C++ allocators designed for high performance in sequential and/or multi-threaded environment (Intel TBB, EASTL, or others).
3) Get familiar with existing implementations of C++ data structures designed for high performance in sequential and/or multi-threaded environment (Intel TBB, EASTL, facebook/folly, Boost, or others).
4) Analyze the usage of data structures in LSU3shell source code and propose their modification for higher program performance.
5) Analyze the possibilities of the usage of vectorization in LSU3shell source code and propose their application for higher program performance.
6) Implement propose changes and measure their impact on performance and memory utilization.

## References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 10, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

# Performance analysis of the `LSU3shell` program

***Bc. Martin Kočička***

Department of Theoretical Computer Science
Supervisor: Ing. Daniel Langr, Ph.D.

January 10, 2019

# Acknowledgements

I would like to thank Daniel Langr and Tomáš Dytrych for their detailed consultations throughout the process of writing of this thesis. I would also like to thank my family for their continuous support.

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on January 10, 2019 . . . . . . . . . . . . . . . . . . . . . .

## Citation of this thesis

Kočička, Martin. *Performance analysis of the* `LSU3shell` *program.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

# Abstrakt

*Ab initio* přístup ke zkoumání struktury atomových jader je na popředí současného vývoje nukleární fyziky. `LSU3shell` je implementací *ab initio* metody zvané *symmetry-adapted no-core shell model* (SA-NCSM) pro vysoce paralelní výpočetní systémy.

Cílem této práce bylo zanalyzovat výkonnové charakteristiky programu `LSU3shell` se zaměřením převážně na dynamickou alokaci paměti, provést rešerši metod a řešení která by mohla vést ke zlepšení výkonu a využití paměti, a provést následnou implementaci.

Tento přístup se ukázal být správný, a bylo možné provést mnoho optimalizací vztahujících se k dynamické alokaci paměti. Výsledkem této práce je průměrné zrychlení `LSU3shell` o 41 %, což nám ušetří až 1,4 milionu core-hodin z naší alokace na superpočítači BlueWaters.

**Klíčová slova** HPC, distribuované výpočetní systémy, dynamická alokace paměti, výkonová optimalizace

# Abstract

*Ab initio* approaches to nuclear structure exploration are at the forefront of current nuclear physics research. `LSU3shell` is an implementation of the *ab* *initio* method called *symmetry-adapted no-core shell model* (SA-NCSM) optimized for distributed HPC systems.

The goal of this thesis was the analysis of the `LSU3shell` program with focus primarily on dynamic memory allocation, research of methods that can be used to improve performance and memory usage, and their application on `LSU3shell`.

The focus on dynamic memory allocation proved to be the right one, leading to many possible optimizations. I was able to reduce the run time by 41 % on average, thus potentially saving up to 1.4 million core-hours of our total BlueWaters allocation.

**Keywords** HPC, dynamic memory allocation, performance optimization

# Contents

# List of Figures

# List of Tables

# Introduction

*Ab initio* models are trying to describe the nuclear structure and reactions starting from fundamental forces among nucleons [2]. `LSU3shell` implements the *symmetry-adapted no-core shell model*, which takes advantage of symmetries inherent to nuclear dynamics, leading to the ability to deal with heavier nuclei than other *ab initio* methods. This research is of importance not only to nuclear physicists, but also to other areas like nuclear energy research. Astrophysicists need to study reactions with unstable isotopes that are impossible to be measured in the laboratory. SA-NCSM can help us understand the processes happening in extreme environments, from stellar explosions to the interior of nuclear reactors.

*Ab initio* methods describe the nuclear structure by solving a many-nucleon non-relativistic Schrödinger equation with interactions among nucleons as the only input. The process that is used to find the solution to the Schrödinger equations is what differentiates various *ab initio* models the most.

SA-NCSM solves this equation by finding eigenstates and eigenvalues of the Hamiltonian, which is computed in a many-nucleon basis that spans the relevant subspace of the Hilbert space, as determined by the symmetry considerations. The Hilbert space is referred to as the *model space*.

The improved `LSU3shell` algorithm as described by Langr *et al.* [3] is divided into three phases. First, many-nucleon basis that spans the given model space is generated. Second, the Hamiltonian is constructed in this basis. And third, the Lanczos algorithm is used to compute the eigenstates and eigenvalues of the Hamiltonian.

`LSU3shell` uses the Message Passing Interface (MPI) library for distributing the calculations and communicating over computational nodes of a supercomputer. On a single node, the Open Multi-Processing (OpenMP) library is used to parallelize local computations using threads. In the current implementation, the MPI load balancing is very simple since we can accurately divide the Hamiltonian into computationally very similar chunks.

Since the memory consumption was a known bottleneck when I joined the project, and the team also already recognized the memory was a performance bottleneck too when switching from GNU C Library userland allocator to tbbmalloc, we decided to focus this work on memory in general, and more specifically on dynamic memory allocation optimizations.

Chapter 1 provides an overview of the best tools that can be currently used to analyze the performance of a program. These tools are applied in chapter 2 to analyze the performance and find the hotspots of the `LSU3shell` program.

Extensive research spans the chapters 3 to 7. Chapter 3 explores the problem of userland memory allocation and follows with detailed research of inner workings of the four most popular userland allocators today. Chapter 4 examines the allocator model in the C++ programming language and looks at popular implementations of C++ STL allocators. Chapter 5 explores the small size optimization and goes over data structures that implement this optimization. Chapter 6 looks at the memory management technique of memory pooling and gives an overview of existing implementations of memory pools. Chapter 7 gives an overview of the hash table data structure and studies existing concurrent solutions.

This research is applied in chapter 8 and the results are given and discussed.

`LSU3shell` as a whole is a result of team effort and collaboration. I will be using the first person pronoun “I” when referring to my own research and experimental work, and the plural “we” will be used when talking about work that was done as part of a team effort.

Chapter **1**

# Analysis tools

Proper analysis should precede any attempts at optimization. When optimizing an application, it is critical to identify the bottlenecks correctly. Even if we can make a function thousand times faster, it does not matter if only 0.0001 % of the total CPU time is spent in said function. The code path on which is spent the most time is called the *hot path*, and the functions in which the most time is spent are called *hot functions*. In this chapter I will introduce a selection of tools that can be used to profile the performance of an application.

## 1.1 perf

`perf`[1] is a powerful performance analysis tool that has been part of the Linux mainline kernel since version 2.6.31 [4]. `perf` uses the Processor Monitoring Counters (PMC) that are recorded by Processor Monitoring Unit (PMU). Brendan Gregg gathered extensive usage examples [5] of all the `perf` tools.

`perf stat` can be used to run a command and gather performance counter statistics. These include cache hits and misses, TLB performance, branch predictor performance, major and minor page faults, CPU migrations, information about the instruction pipeline, and others. Full list of event types that can be gathered can be seen by calling `perf list`.

`perf record` is a sampling profiler. It can either monitor the whole run of a program, or it can be attached to a running process. `perf report` is then used to visualize the gathered data. By default, `perf report` does not show call chains, but it can be force via the `-g` command line flag.

Sampling period for `perf record` can be set through command line parameter `--count=period`. Since `perf record` generates a large amount of data, I had to make the sampling period bigger for some workloads.

[1] `https://perf.wiki.kernel.org/index.php/Main_Page`

`perf annotate` tries to make `perf report` output more understandable—it shows the actual code annotated with data from the profiler, it colors the hot lines, can jump through just the hotspots, and much more.

`perf sched` can be used to trace, measure, and observe the scheduler behavior.

## 1.2 Heaptrack

Heaptrack[2] is a heap memory profile that is available only on the Linux platform. It tracks memory consumption, the number of allocations and deallocations, temporary allocations, and leaked allocations. It show function-by-function summaries, but it can also point to actual line where allocations happen. Similarly to userland allocators described in chapter 3, Heaptrack injects its own `malloc` implementation using `LD_PRELOAD` environment variable. It can either run a program directly and monitor its whole run, or it can be attached to an already running process.

Heaptrack has fairly high overhead—a workload that runs 7 minutes took 4 hours to analyze and generated 7.26 GiB of data. It is still less overhead than the VTune Amplifier's Memory Consumption Analysis, and Heaptrack proved to be very stable for us, while VTune has issues from time to the. Thus if the time and space is not an issue, it is a great tool for analyzing allocation patterns and memory consumption.

## 1.3 Intel® VTune Amplifier

Intel® VTune Amplifier is a performance analysis tool. It can profile on Linux, Windows, and Android targets, and the data can be visualized and analyzed on Linux, Windows, and macOS. It comes with GUI and a command line interface. It is a paid product, but Intel® offers free licenses for students, educators, and open source developers. It supports a very useful comparison mode, which lets you see what exactly changed performance-wise after applying an optimization. Before 2019, VTune Amplifier needed special sampling drivers for advances analyses, but 2019 version removed this requirement if `perf` is available, making the profile much more user friendly. VTune Amplifier contains many analysis profiles, including:

**Hotspots Analysis**

Runs a sampling profiler that tells us in which functions is spent the most CPU time, and if also measures CPU utilization. Basic sampling analysis can be run directly in user mode. Advanced hotspot analysis works with hardware event-based sampling, and it requires either `perf` or special sampling drivers to be installed.

[2] `https://github.com/KDE/heaptrack`

**HPC Performance Characterization Analysis**
Analysis suitable for computationally-intensive applications. It analyzes floating-point operation efficiency, CPU utilization, time spent fetching data from CPU caches and main memory, and the usage of vectorization. It needs special sampling drivers or `perf`.

**Microarchitecture Exploration Analysis for Hardware Issues**
This analysis focuses on efficient pipeline usage and how much time is spent fetching data from L1 cache, L2 cache, L3 cache, and DRAM.

**Memory Access Analysis for Cache Misses and High Bandwidth Issues**
This analysis can help identify performance issues related to memory access. It measures total number of loads and stores, cache misses and latency, and it can identify NUMA-related problems.

**Memory Consumption Analysis**
As the name suggests, this analysis measures memory consumption by each function. It also records how many allocation requests took place and how much memory was deallocated.

**OpenMP Code Analysis**
This analysis shows which parts on code run serially, if there is a load imbalance (a thread finished and waits for other threads on a barrier), and which parallel loops have too little iteration to properly utilize all threads. It can also give simple estimates of potential performance gain by proper threading utilization.

Intel® VTune Amplifier is the most comprehensive performance analysis suite I encountered. It is great for analyzing smaller programs, but for our use-case it had too much overhead—a 10 minute run of `LSU3shell` generated 20 GiB of profiling data even for simplest analyses. Since our program is very allocation heavy, the Memory Consumption Analysis could not handle our workload, and we had to look for another tool.

## 1.4 Intel® Advisor

The profiler part of Intel® Advisor[3] focuses on vectorization. It also includes a tool for modeling threading designs. The vectorization analysis can help identify high-impact under-optimized loops, it can find what is blocking vectorization is some loops, and it can identify loops that can be safely forced by the compiler to be vectorized.

[3] `https://software.intel.com/en-us/advisor`

## 1.5 Intel® Trace Analyzer and Collector

Intel® Trace Analyzer and Collector[4] is a performance analysis tool focusing on distributed MPI applications. It analyzes communication patterns, load balancing, synchronization bottlenecks, communication hotspots, and others.

## 1.6 Intel® Inspector

Intel® Inspector[5] is a memory and thread debugger. It can identify memory leaks, memory corruption, allocation and deallocation mismatches, illegal memory accesses, reading of uninitialized memory, deadlocks, and data races. It can also find errors in persistent memory, which is an emerging class of memory storage.

## 1.7 Compiler Explorer

Compiler Explorer[6] created by Matt Godbolt is a web-based interactive tool that can be used to inspect assembly output of various compilers. It supports multiple programming languages: C, C++, D, Fortran, Go, Rust, Swift, *etc.* It supports all major C++ compilers: GCC, Clang, Intel® icc, and MSVC, and it supports multiple versions and architectures for each of them.

## 1.8 GNU time

GNU time[7], not to be mistaken with bash command `time`), is a simple utility with almost zero overhead that runs another program and reports its resource usage and other useful information. This information includes:

- User, system, and elapsed time.
- Percent of CPU this job got.
- Average and maximum resident set size (RSS).
- Major and minor page faults.
- Voluntary and involuntary context switches.
- File system inputs and outputs.
- Socket messages sent and received.

[4] `https://software.intel.com/en-us/intel-trace-analyzer`
[5] `https://software.intel.com/en-us/intel-inspector`
[6] `https://godbolt.org`
[7] `https://www.gnu.org/software/time/`

- The number of signals delivered.
- The exit status code.

## 1.9 XRay

XRay [6] is a function call tracing system developed by Google that has almost zero overhead when turned off, and moderate overhead when turned on. It inserts small no-op sleds in function entry and exit points. If XRay is turned on, these no-op sleds are overwritten on runtime with instrumentation code. XRay is implemented in the LLVM compiler infrastructure[8].

## 1.10 Valgrind

Valgrind[9] is a popular software suite that consists of six tools: a memory leak and corruption detector (Memcheck), a profiler with call graph generation (Callgrind), a heap profiler (Massif), cache and branch prediction profiler (Cachegrind), and two thread error detectors (Helgrind and DRD).

Valgrind runs the program in its own virtual machine, the program is never run directly on the host CPU. This brings huge overhead, and makes Valgrind only usable for small proof-of-concept programs.

## 1.11 KCachegrind

KCachegrind[10] is a visualizer for profiling data. It it mainly useful for studying call graphs and time spent in different functions. It uses the same data format as the Callgrind tool from the Valgrind suite. Many profilers have support to convert their data to the Callgrind format, so it can be visualized by KCachegrind.

## 1.12 gperftools

gperftools[11] is a suite of high-performance tools—a `malloc(3)` replacement called tcmalloc, a heap checker, heap profiler, and a CPU profiler. tcmalloc is described in detail in chapter 3.

[8] `https://llvm.org/docs/XRay.html`
[9] `http://valgrind.org`
[10] `https://kcachegrind.github.io`
[11] `https://github.com/gperftools/gperftools`

```
#include <gperftools/profiler.h>

int main() {
    f1();

    ProfilerStart("f2.prof");
    f2();
    ProfilerStop();

    f3();

    ProfilerStart("f4.prof");
    f4();
    ProfilerStop();

    f5();
}
```

Listing 1.1: Instrumentation of the gperftools CPU profiler.

### 1.12.1 CPU Profiler

The sampling CPU profiler provides very bare-bones information compared to tools like VTune Amplifier, but it was the only profiler we have tried that was able to handle our actual workloads. It works well with multi-threaded programs and has extremely low overhead. A workload that takes 204 minutes to finish without profiling is finished in 215 minutes with profiling enabled, generating a 1.3 GiB profile output file. We had problems with running gperftools profiler with older versions of `libunwind`, but upgrading to newest `libunwind` solved the problem.

#### Usage

The profiler can either be used to monitor the whole program run, or the code can be instrumented to only profile selected regions. Listing 1.1 shows this selective profiling—only `f2` and `f4` function calls will be profiled and the results will be stored in separate files. The profiling can also be turned on and off using operating system signals, as shown in Listing 1.2.

The `libprofiler` library needs to be either loaded using `LD_PRELOAD`, or he program to be profiled needs to be compiled against the `libprofiler` library. To activate the profiler, the program has to be run with the `CPUPROFILE` variable set. The profiler has otherwise zero overhead, so it can be safely linked even with production binaries. The `CPUPROFILE` variable points to the file where profiling results will be stored. The profiling results can be analyzed

```
env CPUPROFILE=program.prof CPUPROFILESIGNAL=12 ./program &

# start profiling
killall -12 chrome

# stop profiling
killall -12 chrome
```

Listing 1.2: Turning the gperftools CPU profiling on and off using an operating system signal.

```
gcc -lprofiler -g program.c -o program

env CPUPROFILE=program.prof ./program

pprof --callgrind ./program program.prof >program.callgrind

kcachegrind program.callgrind
```

Listing 1.3: Compilation, profiling, and result analysis done with the gperftools CPU profiler and KCachegrind.

using the `pprof` program, that is a part of gperftools. Google has rewritten the `pprof` program in Go[12], and the original has been deprecated, even though it is still functional. `pprof` is able to convert the profiling result to Callgrind format, so they can be visualized with KCachegrind, which I have found superior to `pprof`. The whole profiling sequence, from compiling, to running and data analysis is shown in Listing 1.3.

### Tuning

Some aspects of the CPU profiler can be tuned using environment variables:

**`CPUPROFILE_FREQUENCY`**
: The frequency of sampling in interrupts per second. By default, the profiler takes 100 samples per second.

**`CPUPROFILE_REALTIME`**
: If this variable is set, `ITIMER_REAL` is used instead of `ITIMER_PROF` in the `getitimer` and `setitimer` system calls. `ITIMER_REAL` counts down in wall clock time, while `ITIMER_PROF` counts down in CPU time consumed by the process.

[12] `https://github.com/google/pprof`

## 1.13 strace

strace[13] is a simple utility that can trace system calls and signals a program makes. It is useful for us because it can be used to trace memory management system calls like `brk`, `sbrk`, `mmap`, `munmap`, `madvise`, and others.

## 1.14 Compiler optimization output

All major C and C++ compilers (GCC, Clang, and Intel® icc) support so-called optimization reports. The reports can contain useful information about optimization passes that have been run. Especially important for us are the loop vectorization passes, *i.e.*, the passes that can turn regular loops to ones using vector instructions. The compiler reports can tell us why the loop could not be transformed to vectorized one, and it can help us understand what can be done. Vectorization in GCC is enabled by the `-ftree-vectorize` (and it is included in `-O3` as well), and the reports can be turned on using the `-ftree-vectorizer-verbose=N` flag, where `N` is the verbosity level[14]. Clang has the unified `-Rpass` interface for the optimization pass reports. `-Rpass=loop-vectorize` shows loops that were successfully vectorized, `-Rpass-missed=loop-vectorize` shows loops that failed the vectorization pass, and `-Rpass-analysis=loop-vectorize` show the actual statements that caused the vectorization pass to fail. Vectorization reports in Intel® icc are enabled via the `-vec-report` flag.

[13] https://strace.io

[14] https://gcc.gnu.org/ml/gcc-patches/2005-01/msg01247.html

Chapter 2

# LSU3shell performance analysis

## 2.1 Testing environment

### 2.1.1 BlueWaters

The main system the `LSU3shell` is running on is the BlueWaters[15] supercomputer. Most of the measurements were done on this supercomputer for that reason. BlueWaters is a Cray hybrid (CPU and GPU) supercomputer located at the University of Illinois in Champaign, Illinois with total peak performance of 13.34 PF. It consists of 22,636 XE nodes (CPU) and 4228 XK nodes (CPU and GPU). The total usable storage is 26.4 PB and the total system memory is 1.382 PB. Even with memory this large, it is still a bottleneck for us. The nodes are connected in a 3D torus, and peak node injection bandwidth is 9.6 GB/s.

Each XE node contains two 64 bit AMD 6276 Interlagos CPUs that are based on the Bulldozer microarchitecture. Each CPU has 8 cores at 2.3 GHz and 16 threads—Bulldozer does not use hyper-threading, but each core contains two integer units and two 128 bit floating-point units, that can be either used separately, or as a one 256 bit floating-point unit. It supports SSE4a and AVX vector instructions. The peak performance of each node is 313.6 GF. Most of the XE nodes have 64 GB of memory with the exception of 96 nodes that have 128 GB of memory.

The XK nodes contain the same model of CPU as XE nodes, but they only contain one CPU. Each XK node has one Nvidia GK110 (K20X) Kepler GPU. This GPU has 2688 cores and 6 GB of memory. The peak double-precision performance of the GPU is 1.31 TF. Most of the XK nodes have 32 GB of memory with the exception of 96 nodes that have 64 GB of memory.

[15] `https://bluewaters.ncsa.illinois.edu`

### 2.1.2 STAR

STAR is a smaller CPU cluster located at Faculty of Information Technology at Czech Technical University in Prague. Its nodes are equipped with fairly new Intel CPUs, and we wanted to see how would the performance profile be different compared to older AMD processors in BlueWaters. It is composed of 24 nodes, each node having 64 GB of memory and two 64 bit Intel® Xeon® E5-2630 v4[16] processors. Each CPU has 10 cores at 2.2 GHz. Even though this CPU supports hyper-threading, the hyper-threading is disabled on star. The CPU supports newer AVX2 vector instructions.

### 2.1.3 RSJ1

RSJ1 is a server located at Faculty of Information Technology at Czech Technical University in Prague. Both BlueWaters and STAR run on fairly old Linux kernels, 3.0.101 and 3.10.0 respectively, both released in 2013, while RSJ1 runs Linux kernel version 4.4.0, released in 2016. Some benchmarks we wanted to run required newer Linux kernel, especially comparisons of newest implementations of the GNU C Library. RSJ1 is equipped with 32 GiB of system memory and two Intel® Xeon® Processor E5-2690[17], each having 8 cores and 16 threads thanks to hyper-threading support.

## 2.2 Selected datasets

We picked 4 datasets for benchmarking that cover many different workloads. Their configuration is shown in Table 2.1. I will be referring to the datasets by their assigned letter, *e.g.*, dataset A, dataset B, and so on.

## 2.3 `LSU3shell` analysis

The initial analysis was done using the Intel® VTune Amplifier and Heaptrack. For measurement of both was used a smaller workload, since typical workloads are not manageable to be profiled with these tools, due to large overhead of both.

`LSU3shell` supports so-called *simulation mode* that is enabled when either one of `NDIAG`, `IDIAG`, or `JDIAG` environment variables is set. When the simulation mode is turned on, the code is only run on one node, but it runs the same chunk of computations as if the program was running over multiple nodes. This means that we do not get the final result, but the program outputs some useful information, like the final size of lookup tables for Wigner

[16] `https://ark.intel.com/products/92981/Intel-Xeon-Processor-E5-2630-v4-25M-Cache-2_20-GHz`

[17] `https://ark.intel.com/products/64596/Intel-Xeon-Processor-E5-2690-20M-Cache-2_90-GHz-8_00-GTs-Intel-QPI`

| | Dataset A |
|---|---|
| **Model space** | 20Ne_Nmax04_08_eps0.0003_v2_JJ0 |
| **Hamiltonian** | V_N3LO_Vcoul_15MeV_Nmax12 |
| NDIAG | 211 |

| | Dataset B |
|---|---|
| **Model space** | 20Ne_Nmax04_08_eps0.0003_v2_JJ4 |
| **Hamiltonian** | V_N3LO_Vcoul_15MeV_Nmax12 |
| NDIAG | 211 |

| | Dataset C |
|---|---|
| **Model space** | 16O_Nmax12_SpSnS000_JJ0 |
| **Hamiltonian** | NNLOopt_Vcoul_17MeV_Nmax12 |
| NDIAG | 211 |

| | Dataset D |
|---|---|
| **Model space** | 21Mg_Nmax10cut_JJ5_v3 |
| **Hamiltonian** | N2LOopt_15MeV_Nmax12 |
| NDIAG | 99 |

Table 2.1: The configuration of datasets used in benchmarks.

coefficients and the memory consumption for various parts of the program. This is an important feature for figuring out how many nodes will be needed to complete a calculation, without actually running the code over and over on many nodes. Known sizes of lookup tables can also help us reduce memory usage. All three variables (*i.e.*, `NDIAG`, `IDIAG`, and `JDIAG`) have to be set when running in simulation mode. `NDIAG` specifies the number of diagonal processes when dividing the Hamiltonian, and it has to be an odd number, because of limitations of the used eigensolver implementation. The total number of processes can then be calculated as $\frac{\texttt{NDIAG}\times(\texttt{NDIAG}+1)}{2}$, which means `NDIAG=211` is used to utilize the whole BlueWaters supercomputer. `IDIAG` and `JDIAG` specify the index of the block of the basis the process will be computing. Diagonal processes (`IDIAG == JDIAG`) only calculate roughly half of the work non-diagonal processes do. All measurements will be done on non-diagonal blocks.

### 2.3.1 Performance analysis

In Figure 2.1, we can see that the serial part of the program run takes less than 1.5 % of elapsed time, thus optimizing the serial part does not make sense at this moment.

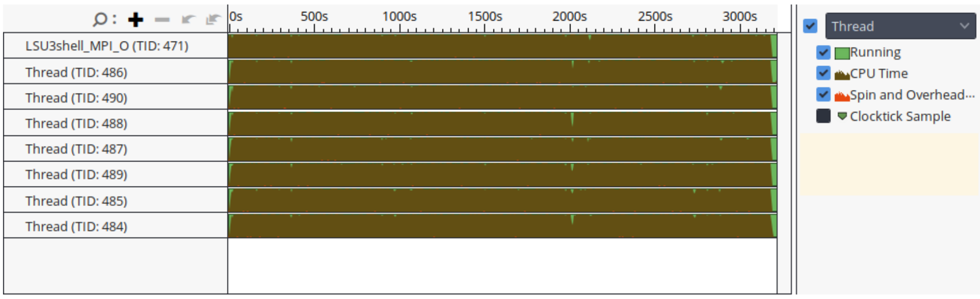
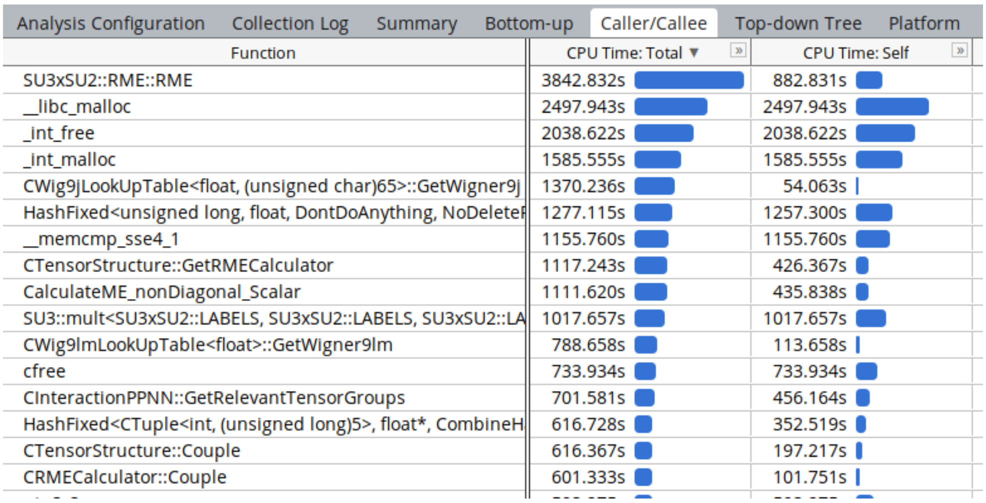



Figure 2.1: CPU utilization of the initial implementation running dataset D with `NDIAG=211` on 8 cores.

From the initial hotspot analysis we can see that 4 out of 5 top hot functions are functions from the userland memory allocator, so there is a lot of space for improvement in this regard. It might be worthwhile to research and try other userland memory allocators.

Other significant hotspot seem to be the lookup tables for Wigner coefficients and also the constructor of the `SU3xSU2::RME` class.




| Function | CPU Time: Total ▼ | CPU Time: Self |
|---|---|---|
| SU3xSU2::RME::RME | 3842.832s | 882.831s |
| __libc_malloc | 2497.943s | 2497.943s |
| _int_free | 2038.622s | 2038.622s |
| _int_malloc | 1585.555s | 1585.555s |
| CWig9jLookUpTable<float, (unsigned char)65>::GetWigner9j | 1370.236s | 54.063s |
| HashFixed<unsigned long, float, DontDoAnything, NoDeleteF | 1277.115s | 1257.300s |
| __memcmp_sse4_1 | 1155.760s | 1155.760s |
| CTensorStructure::GetRMECalculator | 1117.243s | 426.367s |
| CalculateME_nonDiagonal_Scalar | 1111.620s | 435.838s |
| SU3::mult<SU3xSU2::LABELS, SU3xSU2::LABELS, SU3xSU2::LA | 1017.657s | 1017.657s |
| CWig9lmLookUpTable<float>::GetWigner9lm | 788.658s | 113.658s |
| cfree | 733.934s | 733.934s |
| CInteractionPPNN::GetRelevantTensorGroups | 701.581s | 456.164s |
| HashFixed<CTuple<int, (unsigned long)5>, float*, CombineH | 616.728s | 352.519s |
| CTensorStructure::Couple | 616.367s | 197.217s |
| CRMECalculator::Couple | 601.333s | 101.751s |

Figure 2.2: Hotspot analysis of the initial implementation running dataset D with `NDIAG=211` on 8 cores.

### 2.3.2 Memory analysis

Straight away after the first memory analysis, it is clear that the focus on memory we picked as the core of this work was the correct one. The program was calling allocation functions over 230,000 times per second, so over 3 billion allocation calls were made over the 7 minute run time of the program. Almost

500 GiB of memory was allocated over that period. Keep in mind that we usually compute problems that take hours, even tens of hours.

By looking at the distribution of sizes of allocations, shown in Figure 2.3, we can conclude that small size optimization (SSO) might be something worth focusing on, since the vast majority of allocations is of size less than or equal to 64 B. Each color in a bar signifies a location in code where the allocation took place.

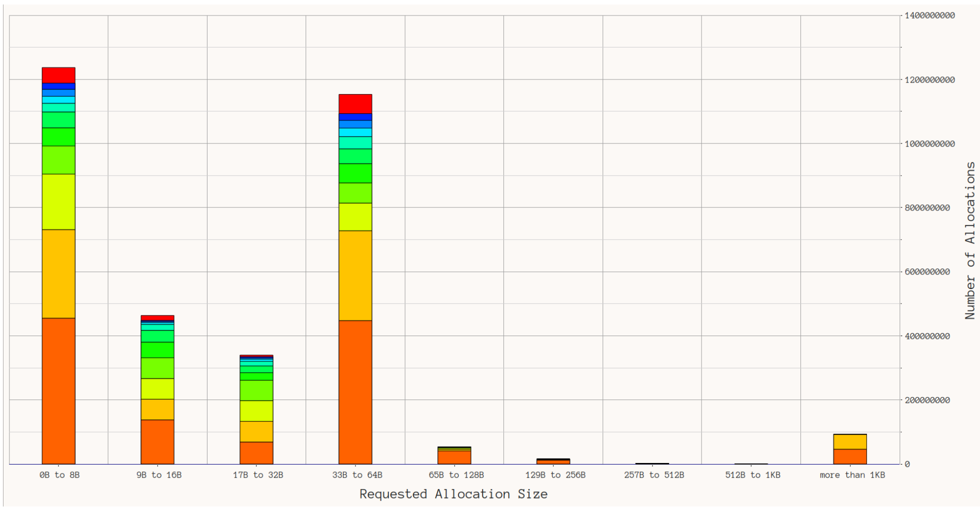



Figure 2.3: Initial allocation counts and sizes.

Looking at the 0 B to 8 B range, the orange bar that is over 30 % of all allocations in this range is a call to the default C++ STL allocator. These come mostly from instances of `std::vector` in our code, which shows that we allocate a lot of really tiny vectors, that would probably gain advantage from using SSO.

Allocating buffers for reduced matrix elements (RME) in the `SU3xSU2::RME` constructor takes up 22 % of allocations in the 0 B to 8 B range. This buffer consists of single-precision floating-point numbers, and since the `float` type that represents these number is 4 B on all of our test machines, this means that all these buffers contain 1 or 2 elements, which again seems like an opportunity to take advantage of SSO. The orange bars in next two ranges are also the allocations of this buffer. The CPU profiling data shown previously confirm that `SU3xSU2::RME` constructor is a hot function.

In the 33 B to 64 B range, the allocations of `SU3xSU2::RME` instances in the `CalculatePNInteractionMeData` function counts for almost 39 % of all allocations. This is a fixed-size allocation, so it is ideal scenario for memory pooling.

In the $> 1$ KiB range is a curious pattern—the yellow and the orange bar have exactly the same size and the number of allocations. After further investigation, this turned out to be a long-forgotten attempt at performance optimization. Both of the allocations are happening in the `CalculateRME_2` function in the `libraries/SU3ME/RME.cpp` file. These are pre-allocated buffers for RME elements. After profiling was done seven years ago, it was discovered that 2 % of the whole `LSU3shell` CPU time was spent calculating the required length of these buffers. It was decided that fixed-size buffers of size 1024 will be allocated for every request, since that should be enough memory for any use case, and it was not causing a radical change in memory consumption.

The first assumption was proven wrong a couple of months ago—we started getting seemingly random segmentation faults on some workloads, and it was exactly because these workloads needed RME buffers larger than 1024. When talking about memory consumption, what matters the most to us is the maximum resident set size (RSS), *i.e.*, the number of pages allocated by this program that are currently backed by physical memory. Maximum RSS truly did not change, even though the total amount of allocated memory went down dramatically (from 490 GiB to 110 GiB). We may conclude from this that these buffers are very short-lived. We decided to revert this change, and the results can be seen on Figure 2.4. Most of the RME buffers still fall into 0 B to 64 B range.

These seem to be the most pressing memory allocation problems right now

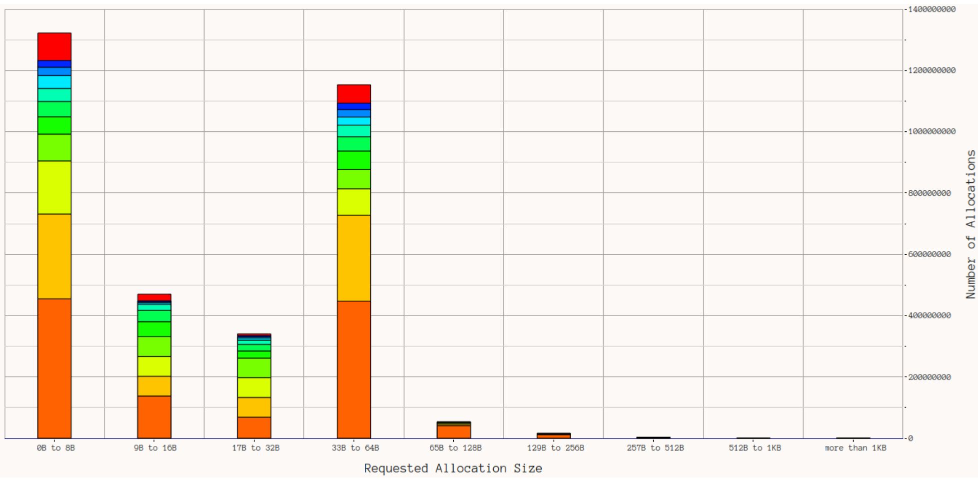



Figure 2.4: Initial allocation counts and sizes after moving back to variable-sized RME buffer in the `CalculateRME_2` function.

When looking at which allocations actually consistently take up the most space, the focus fell on `HashFixed` hash tables that are used to store Wigner coefficients.

### 2.3.3 Hash table

The lookup hash tables for Wigner coefficients are the most used data structure in the whole `LSU3shell`. `LSU3shell` uses its own hash table implementation called `HashFixed`—a fixed-size hash table that terminates the program if the user tries to insert an element and there is no free space available. `HashFixed` has lock-free lookup but insertion requires a lock. It is a chaining hash table, even though the implementation is unusual. It allocates two arrays, `storage` and `bucket`. `storage` is an array of `HashFixed::element` structures that contain the record and index to next bucket, forming a singly linked list. The `storage` array is filled linearly from 0th index, and the `bucket` array stores the mapping from hashed key (modulo size of the table) to the index in the `storage` array. The linking of `HashFixed::element` is used to resolve possible collisions. `HashFixed` has hard-coded limit of holding at most $2^{32}$ elements, because a 32 bit integer is used internally for indexes to the `storage` array. This limit can be simply increased by changing the type to a bigger one.

Our measurements revealed two interesting behaviors:

- Collisions do not happen frequently, and if a collusion occurs, the collision set has in 95% of cases size 2, *i.e.*, the next element in the chain is the right one. Collision sets of size three and larger were very rare, being only 0.24 % of all collision sets. This means open addressing hash table with linear probing might bring a performance gain—we store mostly 18 B keys with 8 B pointers as the value, which means that the two elements would fit on a typical 64 B cache line, making the collision resolution very cheap.
- Lookups are far more frequent than writes. This may be an important factor when designing or picking a new hash table.

`LSU3shell` initially used a `LRUCache` hash table. This table evicts least recently used items, so it can run the same workloads as `HashFixed` with smaller memory footprint. The smaller memory footprint is offset by large performance degradation, so the `HashFixed` is used by default now, even though it is less user friendly. `LRUCache` has similar implementation to `HashFixed`, but it keeps elements linked in a doubly linked list. This leads to need for both insertion and lookup to be locked, and both insertion and lookup share the same lock, which is the biggest reason for the performance degradation.

## 2.4 Conclusion

I know now where the current performance and memory usage hotspots are. The focus of next five chapters will be on technologies and methods that can lesser the impact.

Chapter 3

# Userland memory allocators

Recent profiling done by Google has shown that almost 7 % of all CPU cycles in Google's data centers is spent on dynamic memory allocation [7]. Focusing on optimizing the dynamic memory itself thus seems like a worthwhile activity. Since we are developing a user space application, I will focus on userland (*i.e.*, code running in user space) allocators, more specifically `malloc(3)` replacements. From now on, when talking about dynamic memory allocation, I will be talking about dynamic memory allocation in user space. When talking about operating system specific issues in this chapter, I will be only focusing on Linux kernel, unless stated otherwise. By default I will assume 64 bit system, but I will mention 32 bit variants from time to time.

Dynamic memory allocation priorities changed significantly since 1960s, when the problem was first introduced and researched [8]. Main memory was expensive and scarce, so the foremost objective was decreasing memory usage and fragmentation. As memory sizes grew, and the difference between memory and processor speeds grew larger, the focus shifted to the speed of allocation operations. The trade-off between speed and memory usage is what differentiates many allocators—some focus on being more memory efficient, and some have performance as the main goal. As symmetric multiprocessing (SMP) grew in popularity, the main focus fell on scalability—simply locking the allocator and serializing allocation operations was not enough, and more complex techniques were devised. One of the first papers about allocation on SMP systems was published in 1998 by Larson and Krishnan [9].

An userland allocator keeps track of the heap (and mapped) memory—which parts are allocated, and which are free. Some allocators only care about free memory, since there is no need to keep track of memory that is currently in use by the program.

The user could just allocate memory straight from the operating system by calling `sbkr` or `mmap` system calls, but that would not be efficient, since these system calls incur a context switch, thus being slow. Kernel also allocates memory only in multiples of page sizes, which would have large overhead for small objects. Freed memory is usually not returned by the userland allocator to the operating system instantly, but the memory is instead reused.

Typical call of `malloc` takes approximately 40 instructions and 20 cycles (assuming cache hit) on modern processors [10], so there might not seem to be much space to make significant improvements.

There is also some interest in creating specialized hardware to make dynamic memory allocation faster. Mallacc [10] is an in-core hardware accelerator that speeds up three most important operations of many modern userland allocators: the size class computation, operations with a free list, and memory allocation profiling. Authors of Mallacc claim that they were able to achieve a up to 50 % reduction of `malloc` latency in exchange for 1500 $\mu m^2$ of silicon area, which is less than 0.006 % of typical processor core in 2018. Even though this research is very impressive, hardware accelerators are out of scope of this work.

### Process' virtual address space

Every process has its own virtual address space and addresses are then mapped to physical pages as needed. From the standpoint of the process, it has the whole address space for itself and its threads. A portion of virtual address space is reserved for the kernel in every process, as seen on Figure 3.1. This space is flagged in page tables as exclusive to privileged code, and trying to access it from user mode usually leads to a segmentation fault. Kernel space is usually backed by physical pages at all times. Process' virtual address space consists of multiple segments:

**Text**
: Contains the code (machine language instructions) of the program itself and string literals from the program. It is allocated when the process is created, and it stays the same size for the whole lifetime of the process. This segment is read-only and can be shared among multiple processes.

**Data**
: Contains static data that were initialized by the user.

**BSS**
: Contains static data that were not initialized, and initializes them to zero. Since all of these variables are initialized to zero, they could be stored in the data segment, but to save storage space, only the total size of all uninitialized variables is stored in the executable, and the actual memory is allocated at run time.

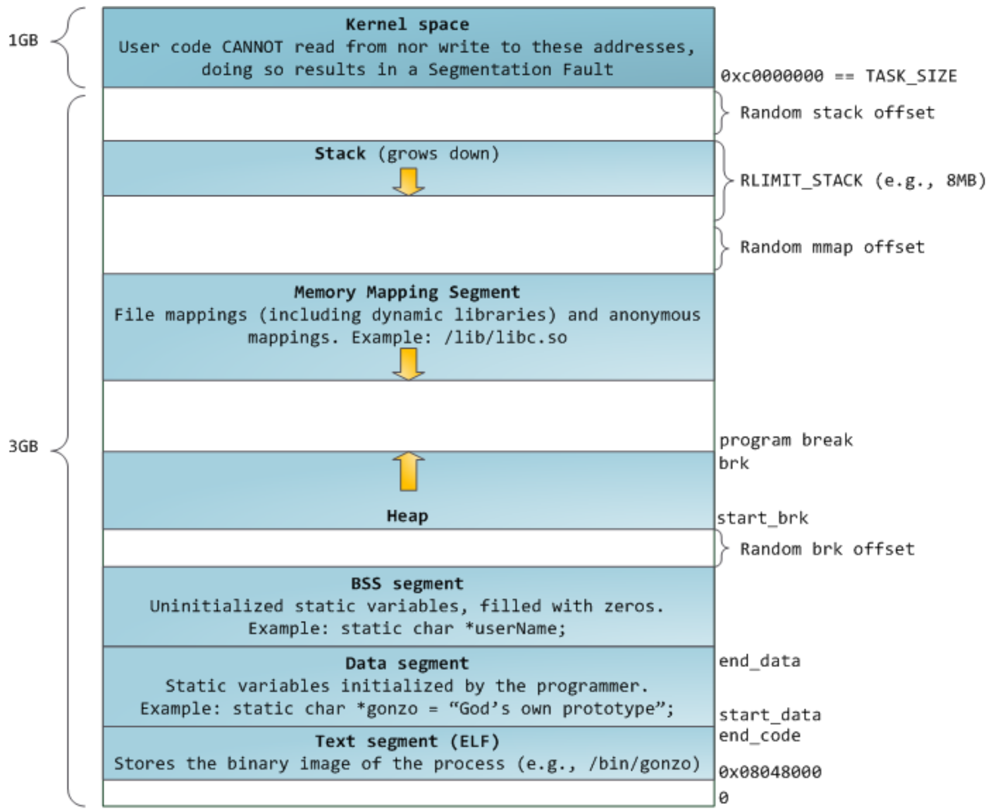
**Stack**
: Contains the process' stack used for local variables and function call stack frames.

**Heap**

This segment is the main interest of this chapter, since this is the memory that userland allocator manages. The top of the heap is called *program break*.

**Memory mapping**

This segment is also used by most userland allocators by utilizing anonymous memory mapping. This segment contains anonymously mapped memory as well as dynamic libraries. Userland allocators are moving more toward using mapped memory, because on 64 bit systems the virtual address space is much larger than the physical memory, and it can reduce fragmentation.




Credit: Gustavo Duarte

Figure 3.1: Typical memory layout of a process on a 32 bit Linux system with ASLR enabled.

Addresses in the stack grow toward smaller addresses, while addresses in the heap grow toward larger addresses. If Address Space Layout Randomization (ASLR) is enabled, starts of the stack, memory mapping, and heap segments have random offset. Memory layout of an object file can be inspected using the `objdump` utility [11].

### 64 bit **architectures**

Some newer allocators (*e.g.*, scalloc [12] and SuperMalloc [13]) only support 64 bit architectures, and take advantage of this knowledge in multiple ways. One is by assuming that virtual address space is many times bigger than the actual physical memory. Therefore there is no need to care about virtual address space fragmentation, and only the pages that are actually used are stored in physical memory via demand paging. It might still be useful to get segmentation fault when accessing virtual addresses that we know should not be accessed, and it can be done using the `mprotect` system call.

Processors from both Intel and AMD have been only using 48 bit for virtual addresses so far, but the new Sunny Cove[18] architecture changes this and extends the addresses to 57 bit. scalloc takes advantage of this and uses the rest of the address to store ABA counters [14].

## Paging

Virtual memory is usually divided into *pages*, which are contiguous (and usually aligned) regions of memory. Typical size for a page is 4 KiB. A virtual page can be backed by actual page of physical memory (called *page frame*), or some file in a storage. This file might be a swap file, or even just a regular file. If the page contains all zeros, it might not have anything backing it at all, and just have a flag stating it only contains zeros [15]. One page frame might have multiple pages of virtual memory corresponding to it. Only a subset of virtual pages is needed to be kept in physical memory—we call this the *resident set*.

When a process tries to access page that is not currently backed by physical memory, a *page fault* occurs. Kernel then has to suspend the process, map the virtual page to a page frame, and resume. This can be very costly, so a good allocator tries to minimize the number of page faults. The process that corrects a page fault is completely transparent to the program accessing memory. For the program, the memory is always the same, it just experiences longer delay when page fault occurs. On some architectures, it is possible [16] to tell the operating system to keep specified pages in memory (*lock* the page), so accessing them never causes a page fault. On the other hand, this can lead to more page faults if we lock too many pages, and there is not enough real pages to sustain other requests. For this reason many operating systems have limit on how many pages can a process lock at a time. Locking pages on POSIX operating systems can be done through `mlock(3P)` and `mlockall(3P)` [17] functions. Even if you lock the page, a *copy-on-write page fault* [18] may

[18]`https://arstechnica.com/gadgets/2018/12/intel-unveils-a-new-architecture-for-2019-sunny-cove/`

| Architecture | Supported page sizes |
|---|---|
| x86-32 [19] | 4 KiB, 2 MiB (PAE), 4 MiB (PSE) |
| x86-64 [19] | 4 KiB, 2 MiB, 1 GiB (pdpe1gb) |
| ARM [20] | 4 KiB, 64 KiB, 1 MiB, 16 MiB |
| ppc64 [21] | 4 KiB, 64 KiB, 16 MiB, 16 GiB |
| UltraSPARC [22] | 8 KiB, 64 KiB, 512 KiB, 4 MiB, 32 MiB, 256 MiB, 2 GiB, 16 GiB |
| IA-64 [23] | 4 KiB, 8 KiB, 64 KiB, 256 KiB, 1 MiB, 4 MiB, 16 MiB, 256 MiB |

Table 3.1: Page sizes supported on most popular CPU architectures.

occur, since operating system can share real pages among many virtual pages if they have the same content. To be sure not to have any page fault occur, it is best to lock the page and write to the memory. The number of page faults is a good metric for benchmarking userland allocators.

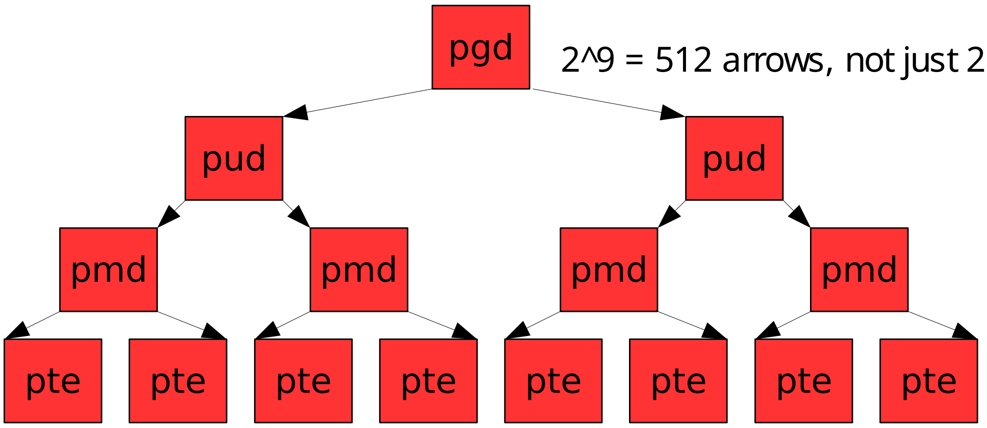
*Page table* is used to store mappings from virtual to physical pages. *Translation lookaside buffer* (TLB) is a hardware cache of these page mappings. TLB is usually very small (tens of entries for first level of TLB), and its effect on performance is significant, so a good allocator should care about TLB behavior.

Since today's systems can have hundreds of gibibytes of memory, but typical page size is still 4 KiB, support for *huge tables* was added. For example, x86-64 supports 2 MiB pages, and some even support 1 GiB pages (processor has to have the `PDPE1GB` flag). This feature is important because the slots in TLB are scarce, and bigger pages mean more memory mapped in the TLB—a 2 MiB huge page takes one spot in TLB, while regular 4 KiB pages would take 512 entries for the same amount of memory. Also 512 page faults can be replaced with just one page fault, but on the other hand the bigger page takes longer to clear and copy, so performance improvements in this regard are not certain. Page tables are usually structured as a tree, and huge page tables only need three levels, while regular pages need four, as seen on Figures 3.2 and 3.3, so lookup is also faster. Additional level of the page table will be needed to handle the larger address space of the Sunny Cove architecture.

There are two ways how to use huge pages on Linux. Either through `hugetlbfs`[19] or by using *transparent huge pages* (THP). Working with `hugetlbfs` can be simplified and tuned using the `libhugetlbfs` [25] library.

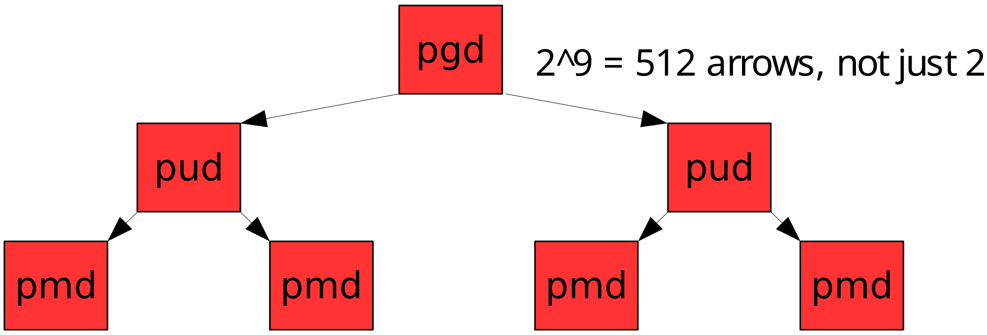
As the name suggest, THP tries to transparently change page sizes according to the need of the application. Currently THP is only supported for anonymous memory mappings, temporary file systems, and shared memory. To reduce memory consumption, it is best to disable huge pages for most processes, and only use them with proper `mmap` and `madvise(MADV_HUGEPAGE)` calls. To be sure that kernel will use huge page for a `mmap` call, the region

[19]`https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt`




Credit: Andrea Arcangeli [24]

Figure 3.2: Page table for x86 architecture with 4 KiB pages on Linux.




Credit: Andrea Arcangeli [24]

Figure 3.3: Page table for x86 architecture with 2 MiB pages on Linux.

should be aligned to a huge page size [26]. This can be achieved, for example, using the `posix_memalign` function. Even though this feature is intriguing, some reports[20] claim that the transparent switching may trigger compaction and defragmentation of pages, which may lead to unacceptably long stalling. This problem has been addressed in the Linux kernel version 4.6 [27] and some aspects of compaction and defragmentation can now be tuned via `/sys/kernel/mm/transparent_hugepage/defrag` [24].

### System calls

Userland allocators work with heap and memory mapping segments. On POSIX-based [17] systems, the `brk`, `sbrk`, and `mmap` system calls are used

[20] `https://groups.google.com/d/topic/mechanical-sympathy/sljzehnCNZU/discussion`

to acquire memory from these segments.

I will refer to both `brk` and `sbrk` as `brk`, since `sbrk` can be implemented as a simple wrapper around `brk`, and they serve the same function. `brk` is used to manipulate the heap segment by setting the program break. `brk(0)` returns the current program break, and we can grow the heap by passing a larger address as a parameter, or shrink it by passing a lower one. `brk` system calls were removed from the POSIX standard in POSIX.1-2001 [28] because they rely too much on the process' memory layout. Most of the POSIX-based systems still implement these system calls: macOS 10.14.2, Linux 4.20.0, OpenBSD 6.4, and FreeBSD 12.0, which are the latest version of these operating systems as of writing this work, all contain the `brk` system call. There is maximum allowed size of heap segment which can be changes using the `setrlimit(RLIMIT_DATA, limit)` system call.

`mmap` is used in userland allocators to create a private anonymous mapping segment. `mmap` can also be used to map contents of a file to a region of virtual address space, but this feature is not needed by userland allocators. Setting the `MAP_PRIVATE` flag ensures that changes will not be propagated to the underlying memory and will be seen only by the calling process. The `MAP_ANONYMOUS` flag specifies that the mapping is not backed by an actual file, and the contents of this memory are initialized to zero. This zeroing out of memory might lead to performance degradation if `mmap` is used excessively. Complementary to `mmap`, `munmap` is used to remove memory mappings. Memory mapping of a process can be examined by looking in the `/proc/{pid}/maps` file or by using the `pmap(1)` utility.

POSIX function `posix_madvise` (usually referred to as simply `madvise` [29]) is used to advise the kernel how to work with specific virtual pages. Some interesting options include:

**`MADV_RANDOM`**
: We expect that these pages will be accessed randomly, so a lookahead buffer may not be beneficial.

**`MADV_SEQUENTIAL`**
: On the contrary to `MADV_RANDOM`, we expect to read the pages sequentially, so a lookahead buffer can be used to increase performance. The system can also free the pages right after they have been read.

**`MADV_WILLNEED`**
: We expect to use these pages in the near future.

**`MADV_DONTNEED`**
: We do not expect these pages in the near future, and the operating system might reclaim them. If a process uses the returned page before it is reclaimed by the operating system, the effects of `madvise` are

overturned. Using pages that have been already reclaimed by the operating system is faster than allocating new ones using `mmap`. Even though `MADV_DONTNEED` does not have to take effect right away (the operating system can decide to free these pages later on), the maximum resident set size is decreased instantly.

**`MADV_REMOVE`** (Linux-specific)
: The pages will be freed, and subsequent accesses will only see memory filled with zeros.

**`MADV_HUGEPAGE`** (Linux-specific)
: Enable Transparent Huge Pages (THP) for specified pages.

### The `malloc(3)` function

Userland memory allocators usually work in two ways—by providing their own functions for memory allocation, and by replacing the default `malloc(3)` implementation (and some even replace the global C++ `operator new` and related functions). The `malloc` function is standardized in both POSIX [17] and ISO C [30, 7.22.3.4] standards. The API is very simple:

```c
void *malloc(size_t size);
void free(void *ptr);
```

The `malloc` function is used to allocate memory from the heap. It either returns a pointer to newly allocated memory or `NULL`, if error occurred (most likely insufficient memory space available). This memory will be always aligned suitably for any C data type. The `free` function is used to signal to the allocator that the memory can be reclaimed and reused, or returned to the operating system.

### Locality of reference

Locality of reference [31] is important kind of behavior with regards to memory. It is usually divided into two types:

**Spatial locality**
: If a memory location is accessed, adjacent memory is going to be accessed in near the future.

**Temporal locality**
: If a memory location is accessed, it will be accessed in the near future again.

Improving locality usually leads to performance improvements, therefore it is one of the goals of some userland memory allocators.

### False sharing

False sharing occurs when two threads read and manipulate memory residing on the same cache line. This scenario can lead to caches being invalidated all the time, thus causing performance degradation. Reducing false sharing is another goal of userland memory allocators.

### Memory blowup

One of the biggest problem in concurrent allocator design is scalable handling of remote frees (*i.e.*, freeing memory allocated by another thread). If allocator can't reuse this memory effectively, it can lead to *memory blowup*. Memory blowup occurs when a program requests significantly larger amounts of memory from the operating system than it actually needs for its computations [32]. Some allocators can not move memory between thread-local data structures, which can lead to memory blowup. Consider following scenario: one thread does significant amount of allocations, while other threads do the deallocations. If the allocator can not move the freed blocks among the thread structures, the other threads will accumulate the free blocks, while the first thread will have to access the central data structure to get new free blocks, thus leading to memory blowup.

### Building blocks

In this chapter I will describe in detail the most popular userland allocators of today—GNU C Library's malloc, tcmalloc, jemalloc, and tbbmalloc. These allocators share similar building blocks. There is usually a structure that serves as a cache for each thread that can be only accessed by that thread, and allocating and deallocating from it does not require locking (even though filling and truncating it might). Then there are underlying structures that might require locking, but the allocators try to mitigate contention by using multiple instances of these structures per CPU core, fine-grained locking, using multiple levels of these structures, and other methods. The allocators usually treat small and large allocations differently, and they also utilize size classes—selected set of sizes to which the request size is rounded up to. Size classes were first used by Tadman in his master's thesis [33]. Size classes are calculated to ensure a desirable balance between fragmentation, latency, and memory usage. Some allocators also make use of *free lists*, which are lists of blocks of memory that are available for allocation. The space reserved for user data is used to store the link pointers, leading to less overhead. Usually these free lists are segregated by size, which means that a free list contains blocks of approximately the same size.

### Usage

There are multiple ways how to use custom userland memory allocators. The simplest one is by replacing `malloc` and related functions by using the `LD_PRELOAD` environment variable. The libraries that are specified in the `LD_PRELOAD` variable will have precedence before any other library, so if the library contains `malloc`, this `malloc` implementation will replace the one that would be loaded by the GNU C Library. For example, if jemalloc is installed at `/usr/lib/libjemalloc.so`, running a program `test` with the default userland memory allocator replaced by jemalloc would be done by following command:

```
env LD_PRELOAD=/usr/lib/libjemalloc.so ./test
```

One big advantage of this approach is that the program does not need to be recompiled, and it can be used even with third-party closed-source software.

The userland memory allocator can be also changed by linking the library when compiling the program. For GCC, tcmalloc installed in standard location, and `program.c`, the linking could be done as follows:

```
gcc -ltcmalloc program.c -o program
```

Some userland memory allocators also provide their own functions for allocation and deallocation. For example, tcmalloc contains all the standard allocation functions with prefix `tc_`, *e.g.*, `tc_malloc`, `tc_calloc`, `tc_free`, *etc.* User can include tcmalloc's headers and use these functions directly. This way, the user could use multiple userland allocators for different parts of the program. Extra care would have to be taken of correct `malloc` and `free` call pairing, since `malloc` and `free` mismatch from different userland allocators would lead to undefined behavior, and probably a segmentation fault.

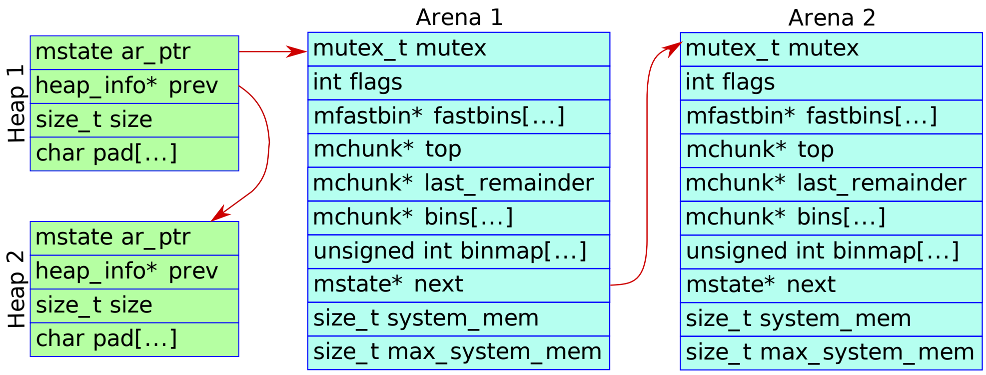
## 3.1 The GNU C Library

Allocator that is shipped with the GNU C Library (glibc) [34] is based on ptmalloc2 [35] by Wolfram Gloger. This allocator is important because it is the default userland allocator on most Linux-based systems [36]. It was created in 2006 as a replacement (and a fork) of the previous allocator dlmalloc [37] created by Doug Lea. dlmalloc was not developed with parallelization in mind—it consisted of only one memory arena, which was locked on allocation, so the allocations were processed serially. As SMP systems were becoming more common, the need for a parallel allocator emerged, leading to creation of this allocator.

Returned memory is by default aligned to $2 \times$ `sizeof(size_t)`, which is usually 16 on 64 bit systems (but by definition the size of `size_t` has no upper bound [30, 6.5.3.4]). When the user tries to allocate 0 B by calling `malloc(0)`, glibc malloc always returns a valid pointer to a chunk of smallest allocatable size, even though it is valid to return `NULL` in this case, as per standard. The GNU C Library's malloc implementation source code is heavily commented and easy to read.

### 3.1.1 Arenas

The main data structure of glibc malloc is *arena* (`struct malloc_state`). An arena contains one or more *heap segments* (`struct heap_info`), which are aligned contiguous regions of memory. These regions are divided into *chunks* (`struct malloc_chunk`), which are used to satisfy allocation requests. Basic structure of an arena is shown in Figure 3.4.




Credit: Josef Kokeš, Tomáš Zahradnický

Figure 3.4: Structure of an arena with multiple heap segments in the glibc allocator.

The first created arena is called *main arena*. Main arena cannot have multiple heaps, and only extends its one heap segment using the `sbrk` system call. All arenas are joined in a linked list through member variables `next` and `next_free`. Heap segments that belong to same arena are also joined in a linked list through member variable `prev`, and they contain pointer to the arena that they are part of. Heap segments are by aligned to either 1 MiB, or 2 × `DEFAULT_MMAP_THRESHOLD_MAX`, if the `DEFAULT_MMAP_THRESHOLD_MAX` variable is defined. `DEFAULT_MMAP_THRESHOLD_MAX` is usually 512 × 1024 on 32 bit systems and 4 × 1024 × 1024 × `sizeof(long)` on 64 bit systems [38].

glibc malloc deals with parallelization by using multiple arenas, which decreases the chance that multiple threads will block each other. The maximum number of arenas is proportional to the number of CPU cores—for 64 bit systems it is 8 arenas per CPU core, and for 32 bit systems it is 2 arenas per CPU core [38]. The number of arenas or the ratio to the number of CPU cores can be configured. If a thread is trying to allocate memory, it goes through all arenas until it finds one that is not locked. If no such arena exists, and we have not hit the arena number limit yet, another arena will be created and assigned to this thread. If the number of arenas is already at the limit, the thread will be added to a waiting queue.

### 3.1.2 Chunks

Unlike all following allocators, glibc malloc stores chunks of different sizes in one heap segment. At the beginning and at the end of the chunk are *boundary tags*, which contain size of the chunk and three flags that are described later in this section. Boundary tags were invented in 1962 by Donald E. Knuth [39]. If the chunk is free, it also contains the information about its size at the end, which makes coalescing of neighboring free chunks trivial and $\mathcal{O}(1)$. All operations maintain the invariant that no two chunks in small and large bins are bordering each other—is such two chunks were to exist, they are immediately coalesced into one chunk in the appropriate operation. Free chunks also contain pointers to previous and next free chunk, forming a free list.

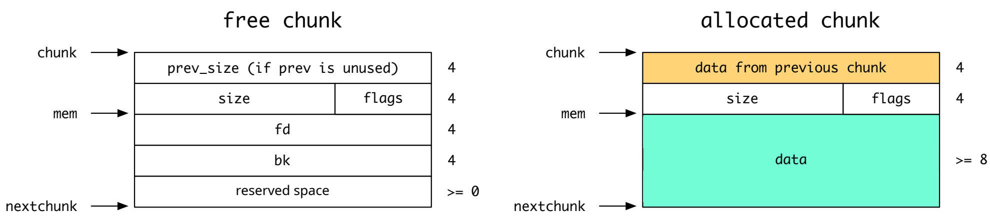
The three flags stored in the boundary tag are:

**`PREV_INUSE`** is set if previous chunk is allocated.

**`NON_MAIN_ARENA`** states if the chunk was obtained from main arena or not.

**`IS_MMAPPED`** specifies if the chunk was requested via `mmap` system call or not. Since chunks obtained by `mmap` are neither in arena, nor are they next to a free chunk, the other flags are ignored, if this one is set.

The structure of free and allocated chunk is shown in Figure 3.5. Since there needs to be space for the boundary tags metadata and at least two pointers inside the chunk, minimum allocated size is 16 B for 32 bit system and 32 B for 64 bit system. The heap allocator cares primarily about free chunks, the chunks that are allocated by the user are not managed in any way until they are freed.



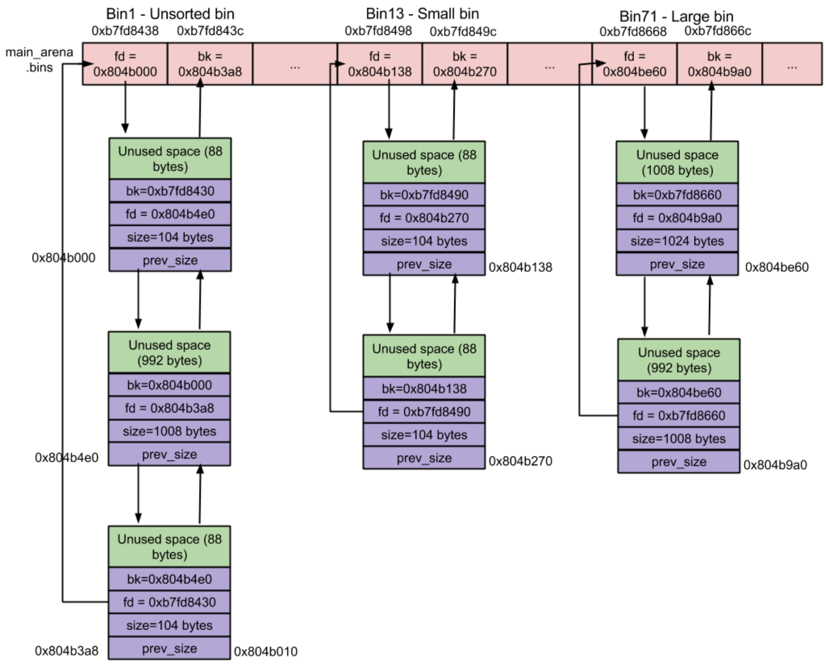

Credit: Vern Paxson

Figure 3.5: Structure of glibc malloc chunk with boundary tags on a 32 bit system.

Since the tags take up 8 B of memory, they can cause significant overhead when allocating a lot of small objects. Allocator metadata in between application data also decreases data locality, since less data fits on one cache line. Boundary tag method also leads to higher internal fragmentation.

### 3.1.3 Bins

Arena also contains *bins* (member variables `bins` and `fastbinsY`). Bins contain *free lists* used to fulfill allocations and deallocations. Free list is a linked list of free chunks. The structure of bins is shown on Figure 3.6. Arena contains multiple bins, where each bin contains free chunks of approximately same size. When allocating, chunks are removed from bins, and while deallocating they are put in.




Credit: `sploitfun.wordpress.com`

Figure 3.6: Bins of the glibc malloc allocator.

In following description of bins, sizes correspond to a 64 bit system with `sizeof(size_t) == 8`. There are 126 bins total, 62 of which are considered *small bins*. Small bins are bins for sizes 32 B to 1008 B and they contain chunks of all exactly the same size. Small bin sizes are spaced 16 B apart, so first small bin contains chunks of size 32 B, second small bin contains chunks of size 48 B, and so on.

All larger bins are called *large bins*, and they are approximately logarithmically spaced, as seen in Table 3.2. Chunks in large bin are ordered by size. Ordering small bins by size is not necessary, since all the chunks in one small bin are of the same size. If more chunks have equal size, the approximately most recently freed chunks are at the front. This first-in first-out allocation order tends to create more consolidated chunks leading to less internal fragmentation. Ordering by size is used for finding least wasteful chunk to use in best-fit allocation. The traversal of ordered list is fast enough so it does not warrant using a more complex ordered data structure.

| **Number of bins** | **Spacing** | **Chunk sizes (usable space)** |
|---|---|---|
| 62 | 16 B | 32 B, 48 B, 64 B, ..., 992 B, 1008 B |
| 32 | 64 B | 1 KiB, 1088 B, 1152 B, ..., 2944 B, 3008 B |
| 16 | 512 B | 3 KiB, 3.5 KiB, 4 KiB, ..., 10 KiB, 10.5 KiB |
| 8 | 4 KiB | 12 KiB, 16 KiB, 20 KiB, ..., 36 KiB, 40 KiB |
| 4 | 32 KiB | 64 KiB, 96 KiB, 128 KiB, 160 KiB |
| 2 | 256 KiB | 256 KiB, 512 KiB |
| 1 | — | up to `DEFAULT_MMAP_THRESHOLD` |
| 1 | unsorted | |

Table 3.2: Number of bins, their spacing, and chunk sizes on 64 bit system.

There is also special *unsorted bin.* Freed chunks are first inserted into the unsorted bin. Chunks that were created as a remainder of the best-fit allocation are also placed into the unsorted bin. Each chunk in unsorted bin has one chance to be used to satisfy an allocation request. If the chunk is not the exact required size, it is moved from unsorted bin to the appropriate bin according to its size.

There is a limit how large a request has to be to be handled by `mmap`. If the `M_MMAP_THRESHOLD` option is set, the threshold is fixed to that value. Otherwise the threshold starts at 128 KiB and is dynamically adjusted according to allocation patterns. The size of a request that will be handled via `mmap` is rounded up to nearest page multiple. Overhead for these chunks is bigger by `sizeof(size_t)`, since we have no following chunk with the `prev_size` field.

Member variable `binmap` contains a bit array of bins, where $i$th bit is set if $i$th bin is definitely empty. This bit array is not always up to date, so it can contain false negatives (the bin is actually empty, but the corresponding bit is not set), but not false positives. `binmap` is used to speed up the traversal of bins by skipping the empty ones.

Chunks of size 16 B to 160 B are called *fast chunks.* Each arena contains 10 *fast bins* (spaced 16 B apart), which are used for holding small chunks. Unlike all previous bins, free lists in fast bins are only singly linked, as seen on Figure 3.7. Chunks are never removed from the middle of the list, so double linking is not necessary. Like small bins, chunks in one fast bin are always of the same size. Chunks in the fast bins are processed in last-in first-out order. The allocator considers all chunks in fast bins as allocated (corresponding `PREV_INUSE` flag is set). They are not coalesced on free, but the consolidation is done in bulk by calling malloc_consolidate. Allocator tries consolidating chunks in fast bins only if a request of size bigger than 64 KiB (FASTBIN_CONSOLIDATION_THRESHOLD) is received. On allocation, if the requested size is in range of fast bins, allocator first tries to satisfy the allocation from the fast bins. Allocator looks in other bins only if there are no chunks in fast bins that are sufficiently large. When free is called on a fast chunk, the chunk is put in the appropriate fast bin. Working with fast bin is lock-free, with the help of atomic CAS instructions. This should make fast bins faster than small and large bins, since those need to be locked using a mutex. Each arena also contains flag have_fastchunks, which is used to skip the fast bins if they are empty. This flag is not up to date at all times. This flag is checked on allocation, and if it is set, chunks in fast bins are consolidated.

*Top chunk* is a chunk that is on top of the heap segment. It does not belong to any bin, and it is used to fill bins when they are empty. It can also be trimmed when the top chunk gets too big.

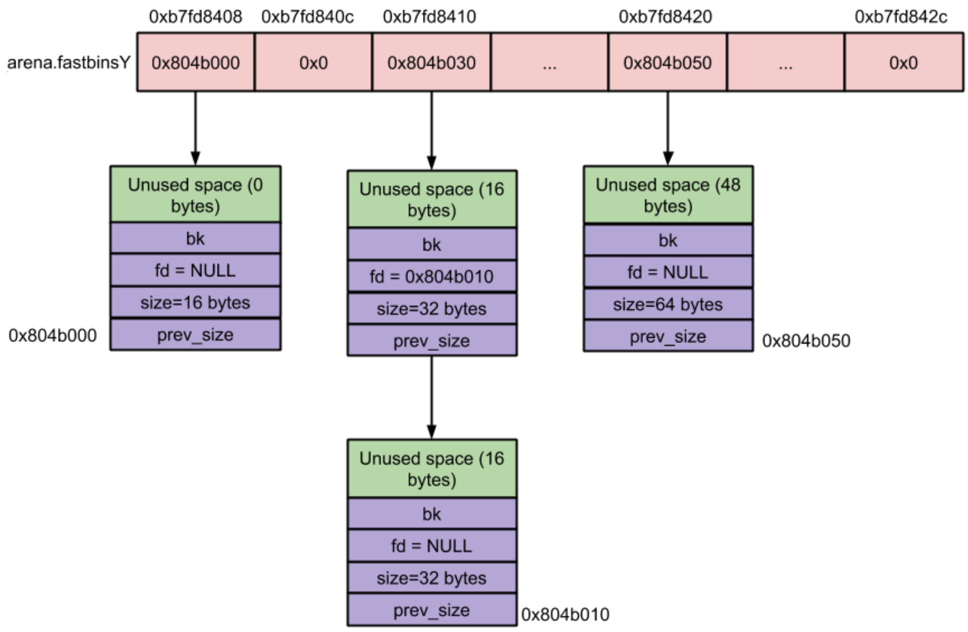
There are two ways how we can get a *remainder chunk*. First is by allocating from unsorted or large bin, when the requested size is less than the size of the selected chunk. Second is by allocating from the top chunk, if top chunk is bigger than the requested size.

Arenas store a *last remainder chunk*, which is a chunk that was created by a split in the most recent small allocation. If a small request can not be satisfied from unsorted or small bins, allocator selects a chunk from the next smallest non-empty large bin by scanning binmap bit array, and even if large bins are empty, the allocator uses the top chunk of the arena. This chunk is then split in two—one satisfying the allocation request, and the second one is now the new last remainder chunk. The reason for storing the last remainder chunk is that when next small allocation occurs, it is satisfied from the last remainder chunk, resulting in better cache locality.

### 3.1.4 Thread-local caches

Thread-local caches were added in glibc 2.26 released on August 2, 2017, after their apparent success in most of other popular malloc(3) implementations (*e.g.*, tcmalloc, jemalloc, tbbmalloc). Thread-local caches improve performance, since there are no locks in neither allocation nor deallocation. Only when we need to fill the cache with empty blocks, we lock the underlying arena. The cache can be filled from all bins—unsorted bin, fast bins, small bins, and large bins. Thread-local cache can be filled without a lock with a chunk that is passed to the free function.

Thread-local cache is yet another set of bins. The size of a request that can be handled through thread-local cache has upper bound, which is by default 516 B for 32 bit systems and 1032 B for 64 bit systems. Every request that is




Credit: `sploitfun.wordpress.com`

Figure 3.7: Fast bin structure.

under this threshold is first tried to be satisfied from thread-local cache. Each thread gets 64 bins, and there can be by default at most 7 items in each bin.

### 3.1.5 Configuration

#### Runtime

Various aspects of glibc malloc can be set at runtime using the `mallopt(3)` [38] function. All of the following options can also be set by an environment variable. Some interesting parameters are:

**`M_MMAP_MAX`**
: Sets the maximum number of requests that can be simultaneously serviced by `mmap`. Setting this value to 0 disables the usage of `mmap`.

**`M_MMAP_THRESHOLD`**
: Specifies a size threshold. If a request is bigger than this size, it is serviced by `mmap`. If this parameter is not set by the user, its default value is 128 KiB, and then it is dynamically adjusted according to the pattern of allocations.

**`M_TOP_PAD`**
: states how much memory is added by every `sbrk` request. It can be

used to lower the amount of system calls. This pad is also retained when releasing memory back to the operating system using `malloc_trim`.

**M_TRIM_THRESHOLD**
: Specifies how much memory has to be in the top chunk to trigger the release of the memory back to the operating system. If this parameter is not set by the user, its default value is 128 KiB, and then it is dynamically adjusted according to the pattern of allocations.

**M_ARENA_TEST**
: Sets how many arenas are created per CPU core. By default this value is 2 for 32 bit systems and 8 for 64 bit systems.

**M_ARENA_MAX**
: Sets the maximum number of arenas that are created.

### Tunables

In addition to `mallopt(3)` and separate environment variables, the GNU C Library also provides *tunables* [40]. Tunables allow to tune a variety of glibc parameters at runtime using the environment variable `GLIBC_TUNABLES` containing a colon-separated list of `key=value` pairs. All the parameters above can be also set by using tunables [41]. For example, to set `M_MMAP_MAX` to 128 KiB and `M_ARENA_TEST` to 4, the `GLIBC_TUNABLES` variable would be set as

```
GLIBC_TUNABLES=glibc.malloc.mmap_max=131072:glibc.malloc.arena_test=4
```

Some additional interesting tunable parameters include:

**glibc.malloc.tcache_max**
: The maximum size of a request in bytes that will be handled through thread-local caches. This value is by default 516 B on 32 bit systems and 1032 B on 64 bit systems.

**glibc.malloc.tcache_count**
: The maximum number of items in one thread-local cache list. By default there can be a maximum of 7 items in each thread-local cache free list. By setting this value to 0, user can effectively disable the thread-local cache.

**glibc.tune.x86_data_cache_size**
: The size of data cache for memory and string functions. This tunable is only available on i386 and x86-64 architectures. It is typically set to L1 size.

#### Compile-time

If the user is willing to recompile the GNU C library, many more parameters can be tuned. These include:

`MALLOC_ALIGNMENT`
: The default alignment of returned memory. The alignment has to be at least $2 \times$ `sizeof(size_t)` and has to be a power of two.

`USE_TCACHE`
: States if the the thread-local caches should be used or not.

These parameters are passed to the compiler, *e.g.*, the default alignment can be changed to `n` by passing `-DMALLOC_ALIGNMENT=n` to the compiler.

## 3.2 tcmalloc

Allocator tcmalloc was created for Google's internal use and it was used in the Chrome web browser until 2014[21]. The *tc* stands for "thread caching", so as the name indicates, each thread holds its own cache for small objects. tcmalloc tries to keep the metadata overhead under 1 % [42]. tcmalloc comes as a part of the gperftools software suite, which also includes simple heap checker[22] and heap profiler[23].

Detailed analysis of tcmalloc's performance (fast path, time spent in different stages of allocation, *etc.*) was done by Kanev *et al.* [10].

### 3.2.1 Pages and spans

tcmalloc divides memory into aligned 8 KiB pages. Multiple contiguous pages are called a *span*.

### 3.2.2 Size classes

Unlike ptmalloc, tcmalloc treats small objects (size $\leq$ 256 KiB), medium objects (256 KiB $<$ size $\leq$ 1 MiB), and large objects (size $>$ 1 MiB) differently. Small allocations can be satisfied from the thread cache, while medium and large objects are always allocated through the central data structures.

Small objects are divided into approximately 88 size classes. The size classes are 8 B, 16 B, 24 B, etc., and the gap between size classes gradually increases. One page always contains objects of only one size class.

When allocating medium and large objects, their size is rounded up to whole pages.

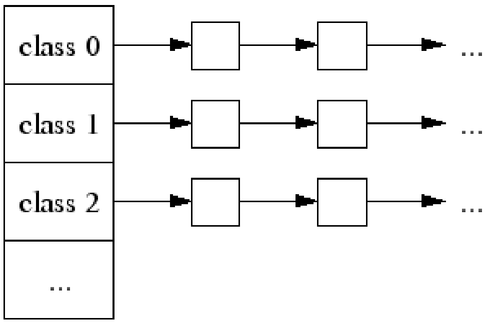
[21] `https://bugs.chromium.org/p/chromium/issues/detail?id=339604`
[22] `https://gperftools.github.io/gperftools/heap_checker.html`
[23] `https://gperftools.github.io/gperftools/heapprofile.html`

### 3.2.3 Thread-local caches

Each thread has its own cache which can fulfill small allocations and deallocations without the need for synchronization. The thread-local cache contains a list of free objects for every size class, as shown in Figure 3.8. Free objects are moved from central free list to thread-local caches as needed. Periodic garbage collection is used to move free objects from thread-local caches back to the central data structures, thus avoiding memory blowup.

The garbage collection is run in two cases. First one is when the size of the cache exceeds specified size (initially 64 KiB by default). This size grows every time garbage collection is run, until hitting specified upper limit for the total thread cache size. If we hit the upper limit, the thread will try to steal memory from other threads' caches.

Second case is when some free list exceeds its maximum length. Thread cache free lists have variable length, which changes with allocations and deallocations from said list. It is important to keep this maximum length appropriate, since a list too short leads to more communication with central free lists (thus increasing contention), and a list too long wastes memory.




Credit: Sanjay Ghemawat

Figure 3.8: Thread cache of the tcmalloc allocator.

### 3.2.4 Central free list

*Central free list* acts as an intermediary between the thread-local caches and the central heap. When allocating small objects, thread first looks in its own cache if it can satisfy the allocation. If not, it asks central free list for the needed free object. If the central free list does not have free object of requested size class, it requests a span from the central heap, splits it into objects, and adds them to the central free list. Then it moves some of these objects to the thread-local cache.

Central free list uses fine-grained locking, where free list of each size class has its own lock. Thus multiple threads can obtain objects from central free list concurrently, if they request objects of different size class.

### 3.2.5 Central page heap

Central page heap contains 128 free lists of spans, where the $k$th free list contains spans of $k$ pages (as seen on Figure 3.9). The whole central heap needs to be locked when accessing, thus increasing contention when accessing frequently. That is why we try to minimize using the central page heap by using the central free list and thread-local caches first.

Each free list actually contains two lists—first for spans that are mapped in current process's address space, and second for lists that have been returned to the operating system (using `madvise(MADV_DONTNEED)`).

Central page heap also contains page map stored in a radix tree. The page map contains mappings from page number to a information about the page stored in `struct Span`. While deallocating, tcmalloc uses this page map to merge free spans, if applicable.

Central page heap fulfills medium object allocations. As stated before, sizes of medium and large objects are rounded up to whole pages. So when allocating medium object of size $k \times 8$ KiB, $k$th free list is used. If there are no free objects in $k$th list, tcmalloc looks into $(k+1)$th list, $(k+2)$th list, and so on, until it finds one with free span. If the span is longer than requested, the span is divided, and the remaining free span is moved back to the free list. If no free list can satisfy the allocation request, tcmalloc first looks in the red-black tree of large objects, and even if it cannot be satisfied from there, it asks the operating system for new memory.

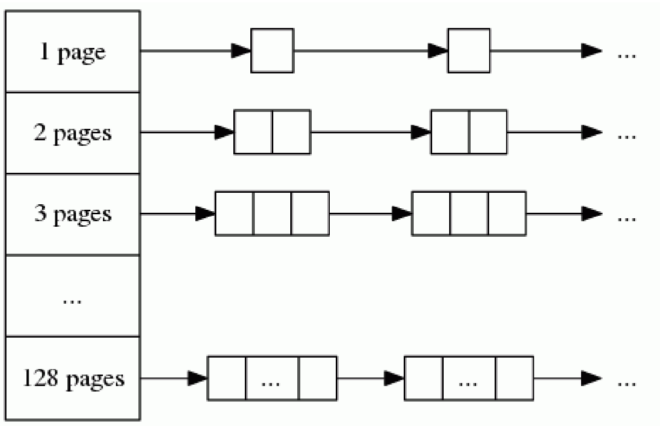
Allocation of large objects is dealt with separately. Spans of free pages that are in total larger than 1 MiB are kept in a red-black tree sorted by size. When doing large allocation, best-fit algorithm is used to find the most suitable span. If the span is larger than requested size, only the needed pages are used, and the rest is moved either to central page heap, or back to the red-black tree. If there is no suitable span in the tree, tcmalloc requests the the memory from the operating system.

Another performance problem with working with the central heap is that a lot of allocation metadata is accessed when working with it, so it may displace application data from CPU caches and TLB, thus leading to higher latency when accessing application data again [43].
The whole architecture of tcmalloc is shown in Figure 3.10.

### 3.2.6 Configuration

Several aspects of tcmalloc (and other parts of the gperftools suite) can be configured. The easiest way of configuration is through environment variables. Some interesting ones include:

`TCMALLOC_RELEASE_RATE`
: Rate at which `madvise(MADV_DONTNEED)` is called to return the memory to the operating system.




Credit: Sanjay Ghemawat

Figure 3.9: Central page heap of the tcmalloc allocator.

**`TCMALLOC_MAX_TOTAL_THREAD_CACHE_BYTES`**
Maximum total size of all thread caches. The default value is 16 MiB, which can be too low when using many threads.

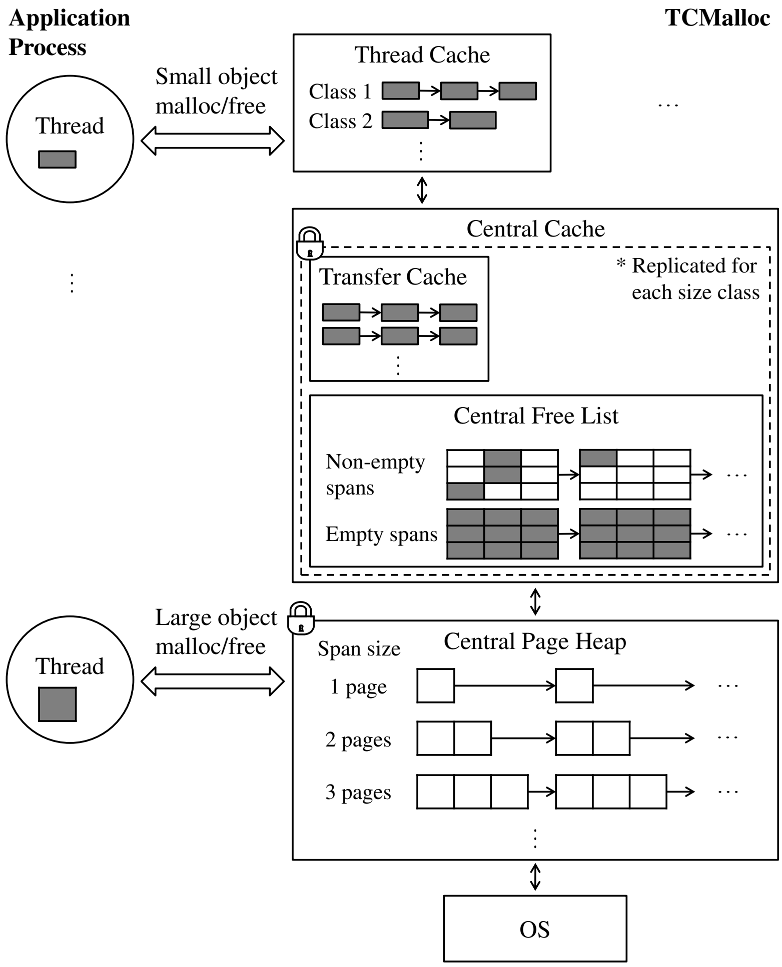
**`TCMALLOC_SKIP_MMAP`**
Do not use `mmap` to acquire memory from the operating system.

**`TCMALLOC_SKIP_SBRK`**
Do not use `sbrk` to acquire memory from the operating system.

## 3.3 jemalloc

Allocator jemalloc was first described by Jason Evans (hence the *je*) in his article "A Scalable Concurrent `malloc(3)` Implementation for FreeBSD" [44]. Evans also provided the initial implementation and the allocator is still being actively developed. The origin story behind this allocator is similar to ptmalloc and dlmalloc—it was an attempt to provide a well parallelized allocator for the FreeBSD operating system, replacing the older and poorly scalable phkmalloc [45]. jemalloc also includes simple heap activity profiling.

It is being used by Facebook in large portion of their infrastructure, it is shipped with the Mozilla Firefox web browser, and it is the default `malloc(3)` in FreeBSD operating system—it is an allocator well tested in production use. The versions shipped with FreeBSD and Firefox are slightly modified compared to the version found in the official GitHub repository [46]. From the three `malloc(3)` replacements discussed in this work (tcmalloc, jemalloc, tbbmalloc), jemalloc seems to be the most popular one right now. One way to measure popularity are "stars" on the GitHub repository, where jemalloc leads, even though both tcmalloc and tbbmalloc are parts of bigger software suites (gperftools and Intel® Threading Building Blocks respectively).




Credit: Sangho Lee, Teresa Johnson, Easwaran Raman

Figure 3.10: High-level architecture of the tcmalloc allocator.

jemalloc focuses on keeping low metadata memory usage, so by design, metadata always take less than 2 % of total memory usage.

### 3.3.1 Arenas

jemalloc combines properties of both previous allocators—it divides allocations into size classes and it is parallelized using multiple arenas. By default four arenas are created for each CPU core. Arena is assigned to a thread at the time it tries to first allocate memory. Arenas are assigned using a round-robin method, which assures that distribution of threads to arenas is approximately uniform. Many older allocators assigned arenas using a hash of a thread identifier, due to lack of thread-local storage, which lead to some arenas being overused, while others were not being used at all. jemalloc falls back to assigning arenas using thread identifier hash only if thread-local storage is not available.

### 3.3.2 Size classes

There are many changes changes in designed since the publishing of the original paper [44]—for example, jemalloc now divides allocations into two categories: small and large, while in the original jemalloc there were three categories: small, large, and huge. Default size classes for 64 bit system, 4 KiB pages, and 16 B quantum are shown in Table 3.3.

| **Category** | **Spacing** | **Size** |
|---|---|---|
| Small | — | 8 B |
| | 16 B | 16 B, 32 B, 48 B, 64 B, 80 B, 96 B, 112 B, 128 B |
| | 32 B | 160 B, 192 B, 224 B, 256 B |
| | 64 B | 320 B, 384 B, 448 B, 512 B |
| | 128 B | 640 B, 768 B, 896 B, 1 KiB |
| | 256 B | 1280 B, 1536 B, 1792 B, 2 KiB |
| | 512 B | 2560 B, 3 KiB, 3584 B, 4 KiB |
| | 1 KiB | 5 KiB, 6 KiB, 7 KiB, 8 KiB |
| | 2 KiB | 10 KiB, 12 KiB, 14 KiB |
| Large | 2 KiB | 16 KiB |
| | 4 KiB | 20 KiB, 24 KiB, 28 KiB, 32 KiB |
| | 8 KiB | 40 KiB, 48 KiB, 54 KiB, 64 KiB |
| | 16 KiB | 80 KiB, 96 KiB, 112 KiB, 128 KiB |
| | 32 KiB | 160 KiB, 192 KiB, 224 KiB, 256 KiB |
| | 64 KiB | 320 KiB, 384 KiB, 448 KiB, 512 KiB |
| | 128 KiB | 640 KiB, 768 KiB, 896 KiB, 1 MiB |
| | 256 KiB | 1280 KiB, 1536 KiB, 1792 KiB, 2 MiB |
| | 512 KiB | 2560 KiB, 3 MiB, 3584 KiB, 4 MiB |
| | 1 MiB | 5 MiB, 6 MiB, 7 MiB, 8 MiB |
| | 2 MiB | 10 MiB, 12 MiB, 14 MiB, 16 MiB |
| | 4 MiB | 20 MiB, 24 MiB, 28 MiB, 32 MiB |
| | 8 MiB | 40 MiB, 48 MiB, 56 MiB, 64 MiB |
| | ... | ... |
| | 512 PiB | 2560 PiB, 3 EiB, 3584 PiB, 4 EiB |
| | 1 EiB | 5 EiB, 6 EiB, 7 EiB |

Table 3.3: Allocation categories and size classes in the jemalloc allocator [1].

Figure 3.11 shows the structure of jemalloc arena (the image is slightly outdated—chunks were renamed to extents and page runs were renamed to slabs, everything else is correct). Arena stores the allocated objects in *extents*, which are continuous blocks of memory aligned to multiples of the page size. How are extents obtained from the operating system can be configured through the `extent_hooks_s` structure. For large allocations, each allocation is backed by its own extents. For small allocations, extents are divided into *slabs*, where each slab can only contain objects of the same size class.

For each size class there is at most one *active* slab per arena at a time. Slab being active means that new objects of its size class are allocated in that slab. For quick allocation and deallocation, arena contains bins, which point to active slabs of different size classes. Bins also contain a list of full slabs and heap of non-full extents. Non-full extents are in a heap because jemalloc always tries to allocate from lowest addresses.

At the beginning of each slab is a bitmap that documents if regions are free or not. This has many advantages over the approach with boundary tags—boundary tags have huge overhead for small objects, and the allocator metadata are interleaved with application data, which can lead to lower data locality, and thus lower performance because of ineffective utilization of CPU caches. jemalloc does not try to prevent false sharing—if users wants to protect against false sharing, they have to take appropriate steps themselves.

When deallocating memory, it does not matter which thread freed the region, and what arena the thread is associated with—freed memory is always returned to the arena from which it was allocated.

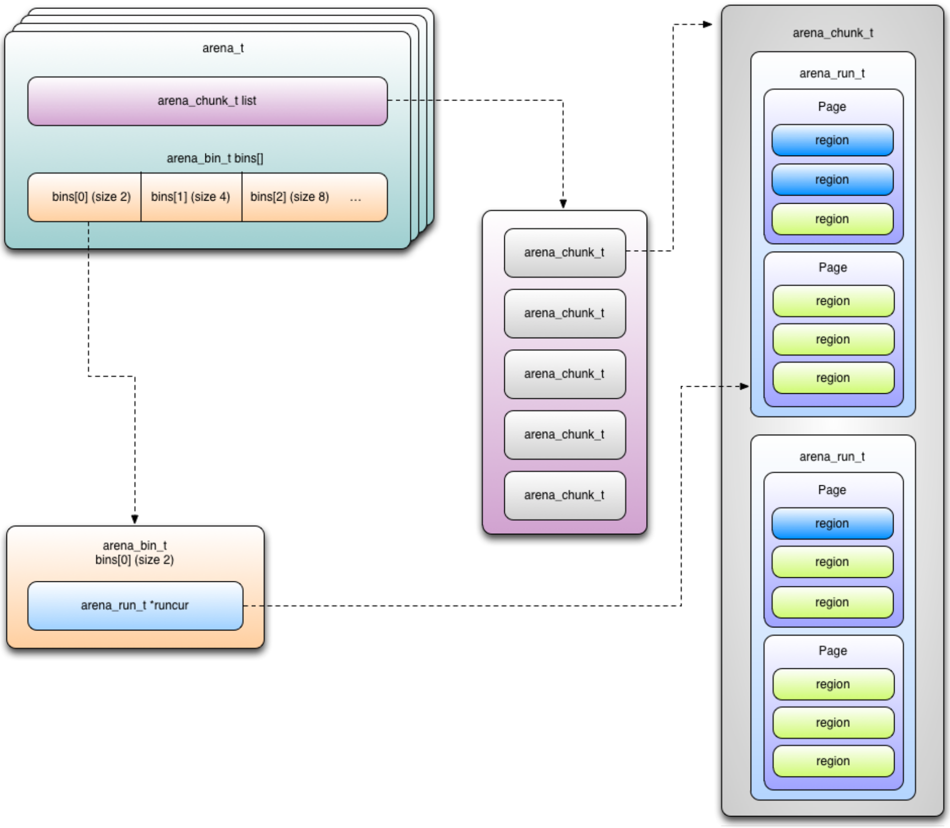
### 3.3.3 Thread-local cache

Initially jemalloc did not have thread-local caches, but when the success of tcmalloc became apparent, thread-local caches were added in addition to arenas. Objects up to a specified size class are stored in the thread-local cache. By default, the maximum size class stored in the thread-local cache is 32 KiB. This can be configured, but the thread-local cache will contain at least all small size classes. Thread-local caches can also be completely disabled.

The thread-local cache structure is very simple, consisting mostly of bins and information for garbage collection. As a time unit for garbage collection is used the number of allocations. Due to periodical garbage collection, *i.e.*, moving free lists from thread-local cache's bins to corresponding arena bins, the memory blowup should not occur.

## 3.4 tbbmalloc

Allocator tbbmalloc [48] is part of the Threading Building Blocks library [49], created and maintained by the Intel Corporation. It is based on work described in the "McRT-Malloc - A Scalable Transactional Memory Allocator" paper by Hudson *et al.* [50]. This is the least popular major allocator of those described




Credit: Patroklos Argyroudis, Chariton Karamitas [47]

Figure 3.11: Structure of an arena in the jemalloc allocator.

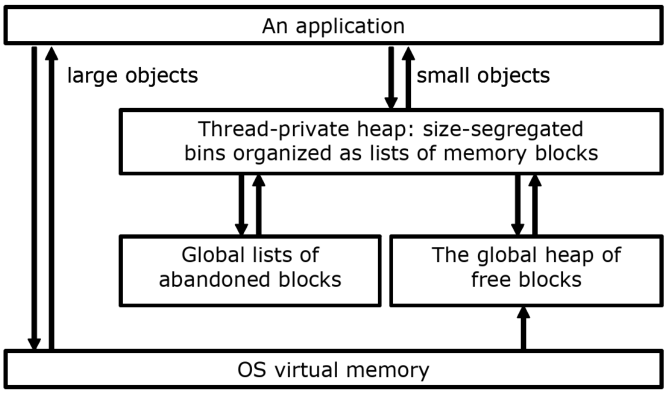
in this chapter, but I included it because it was already used in `LSU3shell` when I joined the project.

tbbmalloc's main focus is on scalability and speed, leading to disadvantages in other areas, to name a few:

- tbbmalloc gives low priority to memory footprint—for example, freed memory allocated for small objects is never returned back to the operating system [13].
- tbbmalloc wastes a lot of memory when allocating objects of size 9 KiB to 12 KiB [51].
- Memory blowup is possible, since the freed objects are always returned to the thread that allocated the memory.
- tbbmalloc does not focus on optimizing and dealing with paging issues.

High-level architecture is shown in Figure 3.12. tbbmalloc divides memory into 16 KiB blocks. These blocks are put inside the global heap of free blocks. Each thread has also its own heap structure, which is how tbbmalloc deals with parallelization, since there is no need to lock the thread heap structure (from now on, I will call this structure a "thread-local cache"). tbbmalloc also introduces size classes, and the thread-local caches are segregated, which means that each block contains only objects of one size class.

When deallocating memory, freed objects are returned to the thread that allocated it, thus allowing the possibility of memory blowup. Each thread has two types of free lists—public (also called foreign) and private. When thread A frees memory of thread B, thread A moves the block to the public list of thread A (thus incurring performance cost of synchronization). Thread always looks to private lists first, and only tries the public one when the private free list can not satisfy the allocation.




Credit: Alexey Kukanov, Michael J. Voss

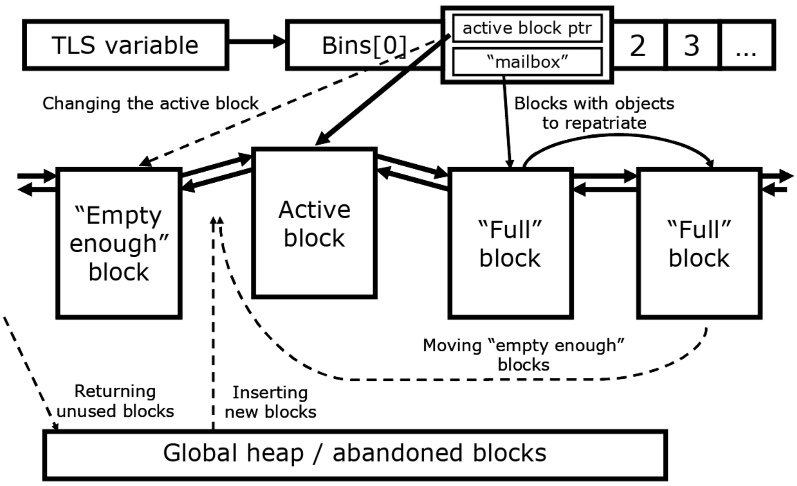
Figure 3.12: High-level architecture of the tbbmalloc allocator.

Structure of thread-local cache is shown on Image 3.13. Local thread heap contains bins which point to active block for given size class. Blocks for one size class are joined into doubly linked list, where full blocks are to the right of the active blocks, and "empty enough" blocks are to the left. When enough memory is deallocated from full block, it is rearranged back to the left of active block. Each block has a header, similarly to jemalloc, and application data are tightly packed together.

## 3.5 Other userland allocators

I have also researched less popular allocators, but extensive research and analysis did not make sense for those. All discussed allocators are focusing on highly parallel workloads.

Hoard[32] is one of the most influential userland memory allocators that are built with concurrency in mind. It focuses on minimizing contention, false




Credit: Alexey Kukanov, Michael J. Voss

Figure 3.13: Thread-local cache of the tbbmalloc allocator.

sharing, and preventing memory blowup.

talloc [52] is a hierarchical memory allocator developed and maintained by the Samba[24] team. It is interesting for its ability of tracking hierarchical data structures and releasing them properly. It can be used non-hierarchically, but the overhead of tracking hierarchies makes it not suitable for non-hierarchical scenarios.

WebKit contains its own allocator called *bmalloc*[25]. It does not override default `malloc` and `free` functions, so I had to write my own wrapper.

TLSF [53] (short for Two-Level Segregated Fit) promises $\mathcal{O}(1)$ `malloc` and `free`, low memory overhead, and low fragmentation.

StreamFlow [54] promises low overhead, high-performance, and better performance due to taking into account and optimizing for locality of reference. It also targets false sharing and tries to improve TLB performance.

rpmalloc [55] is a cross-platform lock-free userland allocator. It heavily utilizes thread-local caches, and many properties of the cache behavior are configurable. It claims to be faster than most of popular userland memory allocators without additional memory overhead.

SuperMalloc [13] was already mentioned in this chapter. It is a userland allocator designed with hardware transactional memory (HTM) and 64 bit virtual address space in mind.

Lockless's LLAlloc [56] is built on top of tcmalloc's design and uses lock-free techniques to minimize latency.

litemalloc[26] is a thread-friendly lock-free userland allocator. It focuses on

[24] `https://talloc.samba.org`
[25] `https://github.com/WebKit/webkit/tree/master/Source/bmalloc/`
[26] `https://github.com/Begun/lockfree-malloc`

minimizing memory fragmentation and it was designed with 64 bit architectures in mind.

scalloc [12] is a scalable general-purpose memory allocator built with 64 bit architectures in mind. Many concurrent operations in this allocator are implemented with lock-free techniques and data structures.

MCMalloc [57] introduces new method ti reduce lock contention by using batch malloc, pseudo free, and fine-grained data-locking.

SSMalloc [58] claims to provide a low-latency locality-conscious allocations with stable scalability. It minimizes `mmap` calls and uses lock-free and mostly wait-free algorithms.

SFMalloc [59] is a lock-free and mostly synchronization-free userland allocator. This allocator neve uses synchronization for common cases, and only uses lock-free synchronization for the uncommon cases.

Chapter **4**

# C++ STL memory allocators

The *allocator* requirements [60, allocator.requirements] are a description of behavior and traits required for memory allocation and deallocation. It is used heavily through-out STL—in containers (except `std::array`), `std::string`, string streams, and others. The class template `std::allocator_traits` [60, allocator.traits] contains a uniform interface to all allocators. The allocator API is simple:

```
T* allocate(std::size_t n)
void deallocate(T* p, std::size_t n)
```

C++ allocators were invented by Alexander Stepanov as a part of the original STL [61]. They were initially meant to solve problems with different pointer types (*e.g.*, near and far pointers) [62], but they are now primarily used to gain performance advantage by managing memory according to specific allocation patterns. The allocator definitions are evolving rapidly, with significant changes in C++11 [63], C++17 [60], and C++20 [64]. Pablo Halpern, one of the main contributors to the allocator model in the standard, stated that only in C++17 allocators became really usable [65]. Before C++11, allocators had to be stateless, making them practically unusable for manually managing memory. More than half of the member types and functions of `std::allocator` were moved to `std::allocator_traits`, and they are deprecated in C++17 and expected to be removed in C++20. Since many copies of the allocator are created when working with STL containers, the allocator should be easily copyable.

The default C++ allocator `std::allocator` [60, default.allocator] is used in STL if the user does not provide a custom one. The `std::allocator` is guaranteed to be thread-safe (except for the destructor) and it is guaranteed to use global `operator new` and `delete` to obtain and release the memory, but it does not define how and when are the operators called.

There were four main problems [66] with C++ allocators in previous standards:

1. Before C++11 custom allocators had to be stateless. An allocator had to always be equal to any other instance of the same allocator class.

2. The allocator is a part of the STL container type. This can lead to two problems—containers with same types but different allocators can't be interchanged easily, and functions and classes have to be templated, if we want them to take containers with any allocator. It is also problem when using interfaces from separately compiled object files. The `std::pmr::polymorphic_allocator` was introduced to solve this problem.

3. If we have nested containers, we might want to pass an allocator to inner containers. Doing this by hand is error-prone, so in C++11 the `std::scoped_allocator_adaptor` [63, allocator.adaptor] was introduced to automatically pass allocators to nested containers. The `std::pmr::polymorphic_allocator` [60, mem.poly.allocator.class] introduced in C++17 is also passed to nested containers.

4. It was assumed that pointer type was always `T*`. Alternate addressing models were supported in C++11 through the `std::pointer_traits` [63, pointer.traits] structure.

All of these problems were addressed in 2005 by Pablo Halpern in his "Towards a Better Allocator Model" [62] C++ standardization paper. There were many C++ standardization papers published [67, 68] trying to improve the flawed allocator model.

There are many reasons one could have to replace the default allocator—higher performance (memory pooling, thread-local heaps, allocating from the stack), debugging of memory allocation errors, or using some special memory, *e.g.*, shared memory, VRAM [69]. Using the default allocator also makes it harder to reason about memory usage, fragmentation, and general behavior of the allocator, unless we know the underlying userland allocator.

## 4.1 Polymorphic memory resources

The `std::pmr` namespace was introduced in C++17 [60, mem.res] to counter many of the previously mentioned problems. The `pmr` stands for *polymorphic memory resource*. Polymorphic memory resources were researched and implemented by Bloomberg in their open-source BSL library [70] over a decade ago, and the final proposal was written by Pablo Halpern [71].

The `std::pmr` namespace introduces many members, including:

**`memory_resource`**
An abstract interface for obtaining and releasing memory.

**`polymorphic_allocator`**
: An allocator that uses underlying `memory_resource` for obtaining and releasing memory. This is the way we can have one allocator type using different memory resources through runtime polymorphism.

**`list, vector, map, ...`**
: Type aliases for all of the standard STL containers (except `std::array`), but with the `std::pmr::polymorphic_allocator` as the allocator.

**`synchronized_pool_resource`**
: Memory resource that requests memory from the userland allocator in chunks, and does the bookkeeping by itself. It owns the memory, so the memory is released when this resource is destroyed, even if `deallocate` wasn't called for some regions. It is thread-safe.

**`unsynchronized_pool_resource`**
: Variant of `synchronized_pool_resource` that is not thread-safe.

**`monotonic_buffer_resource`**
: Memory resource that releases memory only when the resource is destroyed, with `deallocate` being a no-op. Initial buffer can be provided to the resource, so it might use, for example, faster stack memory. If the buffer size is not sufficient, the default allocator is used. It is not thread-safe.

There were attempts [72, 73] in past to standardize other allocators with specific allocation schemes, but so far only the pooling and monotonic buffer got into standard.

## 4.2 Popular implementations

### 4.2.1 Bloomberg BDE

Bloomberg is the company behind many changes to the allocator model in the C++ standard. They implemented the polymorphic allocators in their BSL library [70] more than 12 years before they were standardized in C++17. Lakos *et al.* from Bloomberg released a paper [74] about allocator design including benchmarks. More detailed benchmarks were done by Graham Bleaney [75].

The basic interface and helper functionality is in the `bslma` package, and allocator implementations are in the `bdlma` package. Many of these allocators use memory pools described in chapter 6. Interesting components are:

**`bsls::BlockGrowth::Strategy`**
: In many allocators and pools, the growth strategy can be specified. There are two options: geometric and constant. As the name suggests,

with geometric strategy, the newly allocated memory size grows geometrically, while with constant strategy the new allocated memory size is always the same.

**`bdlma::MemoryBlockDescriptor`**
A value-semantic class that describes a block of memory.

**`bdlma::ManagedAllocator`**
A protocol for allocator that supports the `release` capability, *i.e.*, being able to release all memory allocated by this allocator.

**`bslma::NewDeleteAllocator`**
Simple allocator that uses direct calls to global operators `new` and `delete` to obtain and release memory.

**`bslma::MallocFreeAllocator`**
Simple allocator that uses direct calls to `std::malloc` and `std::free` functions to obtain and release memory.

**`bdlma::AlignedAllocator`**
Interface for memory allocators that support alignment.

**`bdlma::AligningAllocator`**
Wrapper for other allocators that makes sure that allocation have at least the minimum specified alignment.

**`bdlma::SequentialAllocator`**
An allocator that uses `bdlma::SequentialPool` to manage the memory.

**`bdlma::BufferedSequentialAllocator`**
An allocator that uses `bdlma::BufferedSequentialPool` to manage the memory.

**`bdlma::LocalSequentialAllocator`**
Similar to `bdlma::BufferedSequentialAllocator`, but instead of the buffer being supplied by the user, it is allocated by the allocator on stack with compile-time specified size.

**`bdlma::ConcurrentAllocatorAdapter`**
Simple adapter for allocators that are not thread-safe. It takes an underlying allocator and a mutex as parameters, and locks the allocation and deallocation operations. This is very simple synchronization not suitable for high performance applications.

**`bdlma::ConcurrentPoolAllocator`**
Thread-safe allocator that uses `bdlma::ConcurrentPool` as the underlying memory manager. All requests smaller than the pool's block size are handled through this pool, and all other requests are satisfied through

external allocator, which can be either specified on constructor, or the default allocator is used.

**`bdlma::ConcurrentMultipoolAllocator`**
Thread-safe allocator that uses `bdlma::ConcurrentMultipool` as the underlying memory manager.

**`bdlma::MultipoolAllocator`**
Variant of `bdlma::ConcurrentMultipoolAllocator` that is not thread-safe.

**`bslma::TestAllocator`**
Thread-safe allocator adapter that takes a thread-safe allocator as a parameter on construction (or uses the `bslma::MallocFreeAllocator` by default) and accumulates statistics about allocations. The allocator can be set to throw an exception after the total number of allocations goes over the specified threshold. The information this allocator stores include:

- The number of bytes that were allocated by this allocator and are currently in use.
- The total number of bytes that were allocated by this allocator.
- Last allocated and deallocated address.
- The size of last deallocation request.
- The total number of allocation requests.
- The total number of mismatched deallocations, *i.e.*, requests to deallocate memory that was not allocated from this allocator.
- The number of times that pad areas around an allocated block of memory were accessed (only increased on deallocation of such memory block). This measures the number of possible out-of-bounds errors.

**`bdlma::CountingAllocator`**
Simplified version of `bslma::TestAllocator` that only stores stores the number of bytes that are currently in use that were allocated by this allocator, and the total number of bytes that were allocated by this allocator. Unlike `bslma::TestAllocator`, the underlying allocator does not have to be thread-safe. If the underlying allocator is thread-safe, this allocator is also thread-safe.

**`bdlma::GuardingAllocator`**
Thread-safe allocator that can be used to debug memory overflow and underflow. It allocates a read and write protected *guard page* before (or after) the returned allocated block. It is not suitable for production use,

since it has huge memory overhead, and is not that robust. If we are looking for secure allocator, there are better alternatives, *e.g.*, PartitionAlloc[27], which is used in the Chromium and Google Chrome browsers, or the default `malloc(3)` implementation used in OpenBSD [76].

**`bdlma::HeapBypassAllocator`**
: This allocator, as name suggests, bypasses heap memory and allocates directly from virtual memory (*e.g.*, using `mmap` on Linux or `VirtualAlloc` on Windows).

### 4.2.2 Boost

Most of the STL allocators Boost provides are just wrappers around Boost implementations of memory pools, which are described in chapter 6.

#### Boost.Interprocess

Boost.Interprocess[28] library simplifies communication among multiple processes—it provides tools for working with shared memory and memory mapped files, interprocess synchronization primitives, and others.

The `boost::interprocess` namespace contains allocators and related classes, including:

**`allocator`**
: A general purpose allocator that is a wrapper around a *segment manager*. Segment manager is responsible for managing shared memory mapped region or a memory mapped file. This allocator is thread-safe if the underlying segment manager is thread-safe.

**`basic_string, vector, map, ...`**
: Type aliases for Boost's implementations of STL containers that are compatible with the `boost::interprocess::allocator`.

**`slist, flat_set, flat_map, stable_vector, ...`**
: Type aliases for Boost containers that are compatible with the `boost::interprocess::allocator`.

**`node_allocator`**
: Pooling allocator that uses a segment manager. It pools objects of type `T`, and all `node_allocator` instances with same `sizeof(T)` share the memory pool. The pool has a reference counter, and it is destroyed when the last `node_allocator` using this pool is destroyed. This allocator is thread-safe if the underlying segment manager is thread-safe.

[27] https://chromium.googlesource.com/chromium/src/+/master/base/allocator/partition_allocator/PartitionAlloc.md

[28] https://theboostcpplibraries.com/boost.interprocess-managed-shared-memory

**private_node_allocator**

Similar to `node_allocator`, but the pool is not shared among multiple instances. It is not thread-safe.

**cached_node_allocator**

This allocator offers a compromise between `node_allocator` and `private_node_allocator`. It allocates from shared pool like `node_allocator`, but keeps some objects in a private cache that does not need synchronization. It is not thread-safe.

**adaptive_pool**

When using node pool allocators, the memory is never returned to the segment manager—it is only reused by the allocator. The `adaptive_allocator` was introduced to solve this problem. The maximum number of free chunks pool can hold can be set, and if the number of free chunks goes over this threshold, the memory is returned to the segment manager. Adaptive pool allocators have slightly higher memory and performance overhead than node allocators. The memory overhead is used to store metadata. Adaptive pool allocators allocate aligned chunks, so the metadata can be easily accessed using simple binary mask. Otherwise the behavior is similar to `node_allocator`. This allocator is thread-safe if the underlying segment manager is thread-safe.

**private_adaptive_pool**

Similar to `adaptive_pool`, only this allocator owns its pool and does not share it. It is not thread-safe.

**cached_adaptive_pool**

Analogous to `cached_node_allocator`, this is shared adaptive pool allocator that caches memory chunks locally. It is not thread-safe.

### Boost.Pool

The Boost.Pool library offers two STL allocators—`boost::pool_allocator` and `boost::fast_pool_allocator`. Both of these are based on Boost memory pools that are described in chapter 6.

`boost::pool_allocator` is based on `boost::singleton_pool`. This means that all allocators with same `sizeof(T)` share the same pool. This might incur performance degradation in parallel environments due to synchronization, because the singleton pool is naively synchronized using a mutex.

`boost::fast_pool_allocator` is almost the same as `boost::pool_allocator`, but it is optimized for single objects of `T`, even though it can also allocate multiples. `boost::fast_pool_allocator` should be used if we expect to allocate mainly single objects, not multiple contiguous objects.

### 4.2.3 Intel® Threading Building Blocks

Intel® Threading Building Blocks [49] contains two STL allocators: `tbb::scalable_allocator` and `tbb::cache_aligned_allocator`.

#### `tbb::scalable_allocator`

As the name suggests, this allocator is supposed to behave better than the default allocator under parallel workloads. It is only a wrapper around tbbmalloc, which is described in chapter 3.

#### `tbb::cache_aligned_allocator`

In some cases, false sharing can cause vast performance degradation, `tbb::cache_aligned_allocator` tries to avoid this problem. Two objects allocated by this allocator are guaranteed not to be on the same cache line, but this guarantee does not hold if one object is allocated by `tbb::cache_aligned_allocator`, and the other by some different allocator. This allocator pads allocation with additional memory to avoid false sharing, so there is a memory overhead to every allocation, and it can be significant for small allocations. Therefore this allocator should only be used if we know the false sharing is a big problem for our use case. `tbb::cache_aligned_allocator` can also uses tbbmalloc to obtain and release memory by default, but it can use default `malloc` and `free` if tbbmalloc is not available.

### 4.2.4 EASTL

Electronic Arts Standard Template Library (EASTL) [77, 78] is a C++ library created and maintained by Electronic Arts Inc. As the name suggests, it is complementary to C++ STL, containing mostly containers, algorithms, and iterators. EASTL allocators are actually not standard-compliant, so they can't be used interchangeably with STL containers. Unlike standard C++ allocators, EASTL allocators are not bound to any type, and are able to allocate any amount of memory, while standard allocators can only allocate multiples of `sizeof(T)`.

EASTL does not contain any advanced allocator implementations, the only implementation included is a simple wrapper around `malloc`, `free`, and `memalign`.

### 4.2.5 `stack_alloc`

Howard Hinnant's `stack_alloc`[29] is a simple allocator that generalizes the small size optimization described in chapter 5. It allocates stack memory

[29] `https://howardhinnant.github.io/stack_alloc.html`

which size is configured at compile time, and only allocates from heap when the requested size is larger than the remaining stack memory.

### 4.2.6 TP and Medius

TP (short for Two Partitions) and Medius by Jula and Rauchwerger [79] are two STL allocators that try to improve spatial and temporal locality of reference by using hints. These allocators show promising results in the paper, but an implementation is not provided.

Chapter **5**

# Small Size Optimization

Small Size Optimization (SSO, sometimes called Small Buffer Optimization) is an optimization technique that has been gaining popularity in the past few years. The basic idea is that we pre-allocate a small amount of stack memory in advance, and we resort to dynamic memory allocation from the heap only if the stack memory is not sufficiently large. Since allocating memory on the stack is usually just a matter of increasing the stack pointer, it is faster, and the memory is usually hot in cache, leading to better performance due to better locality of reference. This optimization can be, in some cases, made more efficient by instead of allocating separate SSO stack buffer, overlaying existing members of the data structure that are related to the dynamically allocated memory with the stack buffer—a good example of this is `std::string` described in the next section.

## 5.1 `std::string`

One of the best examples of SSO are various `std::string` implementations. Facebook found out that `<string>` is the most included file in their codebase and string functions account for 18 % of all CPU time spend in standard functions [80]. Using their own string implementation with SSO saved them 1 % of CPU time across the whole codebase.

A simplified string implementation is shown in Listing 5.4. The actual string data are allocated on heap, since we need variable run-time size support. Assuming a 8 B pointers and 8 B `std::size_t`, this structure by itself takes at least 24 B. If we were to store the string itself inside this memory, we could theoretically store up to 23 characters. Simpler implementation of explicit SSO is shown in Listing 5.5. This version has simpler implementation, but it carries `SSO_SIZE` bytes of overhead for every string, and that memory is unused if the string is longer than `SSO_SIZE`.

```
class string
{
    char*  data;
    size_t size;
    size_t capacity;
}
```

Listing 5.4: Traditional string memory layout.

```
class string
{
    char*  data;
    size_t size;
    size_t capacity;
    char buffer[SSO_SIZE];
}
```

Listing 5.5: Memory layout of string with explicit SSO.

### `libstdc++` and `libc++`

`libstdc++` is the implementation of C++ standard library that is shipped with the GCC compiler, and `libc++` is the implementation of C++ standard library shipped with LLVM/Clang. The `std::string` implementation in both takes advantage of SSO, but they use a different approach. The following text is true for GCC version 8.2 and Clang version 7.0.0.

`libstdc++`'s `std::string` takes up 32 B, but can only store strings of length up to 15 with SSO. This is because this implementation actually contains explicit SSO buffer, that overlaps with the `capacity` member, but not the `data` member. If SSO is used, `data` then points to the local buffer. This means that the check for SSO is not needed on every operation, leading to less conditional branches.

`libc++`'s `std::string` takes up 24 B, and can store strings with length up to 22 with SSO. This is due to `libc++` overlaying all the members of the structure with the SSO buffer. This means that the check for SSO has to be done on every access of string data, which may lead to performance degradation due to more conditional branching.

## 5.2 Vector implementations with SSO

The `std::vector` can not use SSO because of `std::swap` and invalidation of iterators [60, container.requirements.general]. Since SSO can bring significant performance improvement, many non standard-compliant vector implementations with SSO exist.

Boost's implementation of SSO vector, `boost::container::small_vector`[30], is very simple—it takes the size of SSO buffer as a template parameter, and does not overlay any internal members with SSO data. It simply uses the SSO buffer until the user requests more items than it can contain, and then allocates from dynamic memory. When memory is allocated dynamically, the static buffer is unused, so that vector items can be laid in memory continuously. `small_vector` can be converted to `small_vector_base`, which does not depend on size of the SSO buffer, thus it can be used to avoid templating of client code that uses small vector.

Folly's `folly::small_vector`[31] works the same way as `small_vector` from Boost, but it has more features—by template parameters we can disable the usage of heap all together, and the `size_type` can be also set as a template parameter. `folly::small_vector` unlike `boost::container::small_vector` does not have a size-independent base class, making it little less user friendly. There is a `boost::container::small_vector_base` class, but it serves a different purpose.

LLVM internally heavily uses `llvm::SmallVector`, but it is not easily usable externally as a library, and brings no advantages compared to Folly and Boost implementations.

EASTL provides the `eastl::fixed_vector` as the SSO vector. It is similar to the `folly::small_vector`, the usage of heap can be disabled altogether.

There is a proposal [81] to introduce SSO vector to C++ standard library.

## 5.3 Other data type and data structure implementations

The small size optimization can be used for almost any data type and data structure. `SmallFun`[32] is a SSO version of `std::function` that stores captured variables on the stack. EASTL provides SSO hash table, map, set, list, and others.

Our analysis has shown that we will only benefit from SSO on vectors, so no further analysis of other data structures was done.

## 5.4 `_malloca` and `_freea`

Visual Studio tries a lower-level approach and provides two functions—`_malloca` and `_freea`. As the name suggests, these two functions are analogous to user-

[30] https://www.boost.org/doc/libs/1_69_0/doc/html/boost/container/small_vector.html

[31] https://github.com/facebook/folly/blob/master/folly/docs/small_vector.md

[32] https://github.com/LoopPerfect/smallfunction

land memory allocation functions `malloc` and `free`. The requests smaller than `_ALLOCA_S_THRESHOLD` bytes are served from stack, only larger requests allocate dynamic memory from the heap.

Chapter **6**

# Memory pooling

Even if our userland memory allocator implementation is fast, we can still in some cases improve performance by *memory pooling*. Memory pool requests big regions of memory from the userland allocator, and does the bookkeeping of assigning chunks to the application. Freed memory is not returned to the userland allocator but instead in can be reused, leading to performance gain. Using a memory pool can also reduce fragmentation. Some applications can gain performance by not freeing allocated objects from pool at all, and only free the whole pool when it is no longer needed. This is only suitable for short-lived pools because it can lead to higher memory consumption. Memory pooling can be especially effective if we're only allocating chunks of one size.

## 6.1 Boost.Pool

Boost.Pool library provides multiple memory pool implementations. All of these pools are implemented on top of a memory management algorithm called *simple segregated storage*.

### 6.1.1 Simple segregated storage

Simple segregated storage is represented by the the `simple_segregated_storage` class in the `boost` namespace. Simple segregated storage is responsible for partitioning provided memory region into fixed-size chunks. The simple segregated storage does not provide any alignment guarantees, so the user should keep that in mind if alignment is needed. It keeps a free list of memory chunks. The free list can be in two states—ordered or unordered. The free list is ordered if repeated allocation from the simple segregated storage yields a increasing sequence of pointer values. Corresponding to this are two versions of the deallocation functions—`ordered_free` and `ordered_free_n` keep the list in order, for the price of $\mathcal{O}(n)$ complexity, while `free` and `free_n` are $\mathcal{O}(1)$, but they may break the ordering. In case of Boost.Pool, the user does not have to work with simple segregated storage directly, but it is used by the pools internally.

### `boost::pool`

`boost::pool` is a general implementation of a memory pool on top of `boost::simple_segregated_storage`. All chunks that were allocated from a pool are freed when the pool is destroyed. As with simple segregated storage, pools can also be considered ordered and unordered, depending on if the pool's free lists are ordered or not. Ordered pools are better for allocating multiple contiguous objects, but the deallocation is $\mathcal{O}(n)$. Unordered pools are very fast for single object allocation, but allocating contiguous arrays of objects might be slow and inefficient, since the pool might have enough contiguous memory available, but it does not know about it because the contiguous chunks are not coalesced. Memory chunks returned by `boost::pool` are guaranteed to be properly aligned. User can supply custom allocator that is used to request and release the blocks of memory that are passed to the simple segregated storage.

The chunk size that will be returned by `malloc` and `ordered_malloc` is set in the constructor, and can not be changed during the lifetime of the pool. The size of the first block and the maximum size of the block allocated by the simple segregated storage can be also set via constructor parameters. Ordered `malloc` first tries to allocate from the simple segregated storage, but if it is empty, it allocates new block and coalesces free lists to put the chunks in order. The ordered allocation is still amortized $\mathcal{O}(1)$. Ordered `malloc` has also overload with size parameter `n`, which allocates `n` $\times$ chunk size of contiguous memory. If we're allocating memory for C++ objects, and we want to properly initialize them, the placement new semantics have to be used.

### `boost::object_pool`

`boost::object_pool` is a simple pool that can be used for fast allocation and deallocation of objects. It is very similar to `boost::pool`, but instead of taking the size of the chunk as the constructor parameter, is is templated with type `T`, and the chunk size is `sizeof(T)`.

Since the pool knows the type of the object it is allocating, it provides function `construct` that can be used to allocate and properly initialize the object via its constructor without using the `malloc` and placement new combination. There is also a `destroy` function that properly destroys the object—calls the destructor, and frees the memory in the simple segregated storage.

### `boost::singleton_pool`

`boost::singleton_pool` is a pool that can be shared for types with the same size. Template tag parameter is used to differentiate between different singleton pools. The singleton pool takes the size of the chunks as a template parameter, and a mutex type that is used to synchronize accesses to the underlying pool. If the pool is not to be used in concurrent environment, Boost provides `boost::details::pool::null_mutex`, where the locking is a no-op. The size of the first block and the maximum size of the block allocated that is passed to simple segregated storage can be set via template parameters. `boost::singleton_pool` uses `boost::pool` as the underlying pool. By default, the underlying memory pool is never freed. The `purge_memory` function can be used to release all memory blocks from the underlying pool, and `release_memory` can be used to release memory blocks that do not have any chunks allocated from them.

## 6.2 Bloomberg

Bloomberg's BDE library [70] includes many implementations of memory pools. All of them except the `bdlma::ConcurrentFixedPool` have a corresponding STL allocator that uses the pool as the primary memory resource. These allocators are described in chapter 4. The memory pools provided in BDE are:

**`bdlma::SequentialPool`**
: Fast sequential pool that stores memory blocks in internal dynamically-allocated buffers. It can satisfy requests of varying sizes. This pool is best for single-threaded use when the user does not know the approximate size of memory they will need.

**`bdlma::BufferedSequentialPool`**
: Allocator that allocates sequentially from user-supplied buffer, and when the memory in user-supplied buffer is insufficient, additional buffers are allocated. The user can specify growth rate of the additional dynamically allocated buffers. For best performance, the user know should know approximately how much memory will be needed and the supplied buffer should be memory from the stack. When additional buffers are allocated, the performance decreases significantly.

**`bdlma::ConcurrentPool`**
: Thread-safe pool that manages memory blocks of fixed size specified on construction. Growth strategy of newly allocated memory blocks can be also adjusted on construction. If it is not specified, geometric growth is used. This pool overrides global placement operator `new` and `delete` to make allocating from these pools simple using the placement new semantics. Deallocation is lock-free, while allocation can lock when replenishing the memory block pool.

**`bdlma::Pool`**
Variant of `bdlma::ConcurrentPool` that is not thread-safe.

**`bdlma::ConcurrentFixedPool`**
Similar to `bdlma::ConcurrentPool` with the difference being that this pool has fixed limit of how many memory blocks it can contain. This change makes lock-free allocation possible.

**`bdlma::ConcurrentMultipool`**
Thread-safe memory manager that maintains multiple `bdlma::ConcurrentPool` objects. First pool is for memory blocks of size 8 B, and successive pools are always two times the size of the previous one (*i.e.*, 8 B, 16 B, 32 B, and so on). Allocation requests are rounded up to the closest larger or equal pool size. Maximum number of pools can be set on creation.

**`bdlma::Multipool`**
Variant of `bdlma::ConcurrentMultipool` that is not thread-safe.

## 6.3 nginx

The nginx HTTP server comes with a simple high-performance pool implementation in C. Even though the pool is not released as a separate library, it is easy to use in other projects due to having no external dependencies and the internal dependencies are only architecture-specific configuration options. The nginx pool has very specific behavior and features—small objects can not be returned to the pool, they are only freed when the whole pool is destructed, and objects of variable size can be allocated from the pool. This makes the nginx pool suitable for short-lived local allocations, which is a common use-case in servers. The nginx pool is not thread-safe.

The nginx pool is represented by the `ngx_pool_s` structure. The pool is created using the `ngx_create_pool` function (refer to Listing 6.6 for full API), and it is destroyed by the `ngx_destroy_pool` function call. The pool can be reset using the `ngx_reset_pool`, if the user wants to reuse the pool. `ngx_palloc` is used to allocate aligned memory from the pool, and even though there is a `ngx_pfree` function, it can only be used to free large objects, since small objects can not be freed from nginx pool, as mentioned earlier. `ngx_pnalloc` can be used to allocate from the pool if we do not need aligned memory. When the pool needs to allocate more memory, it allocates a fixed size region, not using geometrical growth that can be used in Bloomberg and Boost pools.

```
ngx_pool_t*  ngx_create_pool(size_t size, ngx_log_t* log)

void         ngx_destroy_pool(ngx_pool_t* pool)

void         ngx_reset_pool(ngx_pool_t* pool)

void*        ngx_palloc(ngx_pool_t* pool, size_t size)

void*        ngx_pnalloc(ngx_pool_t* pool, size_t size)

ngx_int_t    ngx_pfree(ngx_pool_t* pool, void* p)
```

Listing 6.6: The nginx pooling API.

## 6.4 foonathan/memory

The `memory` library by Jonathan Müller contains multiple C++11 compatible allocators and pools. The is only a one pool class, `memory_pool` from the namespace `foonathan::memory`, but it can be divided into three time using a template tag:

**`node_pool`**
: Pool optimized for *nodes*. In this library, a node is a memory region sufficient to hold one single object. It keeps nodes in a free list.

**`array_pool`**
: Pool optimized for arrays of nodes. It keeps the internal free lists ordered, trading better memory usage for performance.

**`small_node_pool`**
: The free list for `node_pool` is a regular linked list that stores the pointer to next element embedded inside the node's memory. This means that every object in `node_pool` has to be at least as big as a pointer. For small objects that can be a big overhead, so `small_node_pool` solves that by only storing 8 bit index, with a little bit more bookkeeping (thus being slower).

The `foonathan::memory::memory_pool` can only allocate objects of one size, or their multiples in case of `array_pool`. If we want to support more sizes, there is a `foonathan::memory::memory_pool_collection` that stores multiple pools of different sizes, and picks the appropriate one. It can either store a separate pool for every size, or it can store pools with node sizes that are powers of two. The maximum node size is set on construction.

Chapter **7**

# Concurrent hash tables

Hash table is a data structure that maps keys to values, and it is a basic building block of many high-performance parallel systems. Even though `LSU3shell` uses many data structures, the only one that has significant performance impact is a concurrent hash table, as seen in the analysis in chapter 2. Hash table is usually backed by an array and a *hash function* is used to derive the index in the array from the key. The elements of the array are called *buckets*, and the key/value pair is called a *record*.

## 7.1 Hash function

One of the most important factors of hash table performance is the selected hash function. The most important property of a good hash function is a uniform distribution of values across buckets. If the hash function is not good, it leads to *collisions*, and the performance of the whole hash table degrades.

### Popular hash functions

SMHasher [82] is a complex benchmark that tests the uniformity of distribution, collisions, and performance of hash functions. The original SMHasher tested ten hash functions, and a fork [83] by Reini Urban added more than 30 additional hash functions. The hash function implementations we tested were picked based on this benchmark.

`boost::hash_combine` from the Boost.ContainerHash library was originally used in `LSU3shell` as the main hash function. `boost::hash_combine` can be called multiple times using the result of previous call as the seed to incrementally build the final hash. The `boost::hash_combine` implementation is simple, consisting of one XOR, two bit shifts, and three additions.

## 7.2 Collision resolution

A collision occurs when the hash function calculates the same index for two different keys. The keys have to be also stored in the table for resolving collisions. There are many ways how to handle collisions—the two most popular are *chaining* and *open addressing*. Chaining hash tables store linked lists in buckets, and the records are chained in the linked list. Open addressing hash tables embed the records inside buckets, and if collision occurs, another bucket is selected.

The process of selecting buckets is called *probing*, and there are many probing algorithms [84, p. 272], *e.g.*, linear probing, quadratic probing, double hashing, cuckoo hashing [85], hopscotch hashing [86], and others. The authors of hopscotch hashing claim it is well suited for concurrent hash tables. There has been a lot of research done [14, 87, 88, 89] on concurrent cuckoo hashing, and the results seem promising.

The *load factor* is a ratio between the number of used buckets and the number of empty buckets. When load factor gets high, hash tables tend to degrade in performance, since higher load factor means more collisions. Load factor is usually used to decide when it is time to resize the hash table.

## 7.3 Popular implementations

### 7.3.1 Threading Building Blocks

**`tbb::concurrent_hash_map`**

A resizable hash table that supports concurrent traversal, search, insertion, and erasure [51, p. 91]. It uses chaining to resolve collisions.

Access to elements is done through *accesors*. Accessor acts as a smart pointer to the record in the hash table. It holds a lock on the record when it is created, and the record in unlocked when the accessor is destroyed, or when the `release` method is called.

**`tbb::concurrent_unordered_map`**

A resizable hash table that is more restricted than `tbb::concurrent_hash_map`—it supports concurrent insertion, search, and traversal, but not erasure. This map might use locking internally, but the locking is never visible to the user.

`tbb::concurrent_unordered_map` uses a simplified split-ordered list [90] as the underlying data structure. Split-ordered list is a concurrent lock-free unordered associate container. Even though `tbb::concurrent_unordered_map` does not support concurrent erasure, in the original paper the split-ordered list supported it. Rehashing is significantly faster [91] compared to `tbb::concurrent_hash_map`. It supports `operator[]`, making it more user-friendly.

### 7.3.2 Folly

#### `folly::AtomicHashMap`

`folly::AtomicHashArray` is a fixed-size lock-free open addressing hash table. It is the basic building block of `folly::AtomicHashMap`.

`folly::AtomicHashMap` [92] is a resizable concurrent unordered hash table built on top of `folly::AtomicHashArray`. As the name suggests, it relies heavily on fast atomic operations for synchronization. It claims to have good memory fragmentation properties and to be 2 to 4 times faster than `tbb::concurrent_hash_map` in highly concurrent environments. To achieve this speed, it has a number of limitations:

- Keys must be a 32 bit or a 64 bit integers. The keys have to be swapped using the CAS atomic operation, and most modern architectures only support 32 bit an 64 bit lock-free atomic CAS operations.

- It can only grow to approximately 18 times the initial capacity, selecting initial capacity is thus very important. Picking initial capacity that is too small might lead to the table not having enough space, and picking initial capacity that is too large might lead to wasteful memory usage.

- There need to be at least three reserved key values—indication of empty, locked, and erased key.

- Memory left by erased records can not be freed or reused.

The probing method can be customized via a template parameter. Folly includes two basic methods—linear probing and quadratic probing. The performance/memory usage trade-off can be tuned by setting maximum load factor in the constructor. When writing a record, the key is locked using CAS atomic operation while the value is written. The `find` function is wait-free.

This table does not support rehashing [93]—when it reaches its maximum load factor, it grows by allocating additional hash tables, leading to performance degradation when the table grows over the initial capacity. If multiple additional hash tables are allocated, they are searched one by one.

#### `folly::AtomicUnorderedInsertMap`

`folly::AtomicUnorderedInsertMap` is a fixed-size chaining hash table that supports lock-free access. Contrary to `folly::AtomicHashMap`, the keys can be arbitrary values. Reading from the table is wait-free, and inserting is lock-free. It has some limitations:

- It is insert-only hash table. Updating can be implemented by the user, but the hash table itself does not support it.

- It can not grow, once the hash table is full, the user will no longer be able to insert records.
- The default maximum capacity is $2^{30}$ records, since the table uses 32 bit indexes internally, and 2 of those bits are used for a flag containing the bucket's state. The type of the index can be changed via a template parameter.

### 7.3.3 Junction

Junction [94] is a library by Jeff Preshing consisting of four concurrent hash tables, three of which are suitable for high-performance environment. In Junction tables, the hash is stored instead of the key, so the hash function has to be invertible for resolving collisions. This differentiates these hash tables from all others I researched. The key has to be an integer or a pointer type.

#### Quiescent state-based memory reclamation

All Junction tables rely on quiescent state-based memory reclamation (QSBR) [95]. QSBR requires more involvement from the user than other techniques. Each participating thread has to periodically call `junction::DefaultQSBR.update` at a moment when it is in a *quiescent state*, *i.e.*, not working with the table.

#### `junction::ConcurrentMap_Linear`

`junction::ConcurrentMap_Linear` is a resizable lock-free open addressing hash table based on Cliff Click's lock-free hash table [96] implemented in Java, which was one the first working lock-free hash tables. It uses linear probing to resolve collisions.

#### `junction::ConcurrentMap_Leapfrog`

A hash table similar to `junction::ConcurrentMap_Linear`, but it uses probing strategy loosely based on hopscotch hashing [97], which should improve efficiency when the load factor is high. It should also scale better.

#### `junction::ConcurrentMap_Grampa`

A hash table similar to `junction::ConcurrentMap_Leapfrog`, except it is gets split into multiple fixed-size `junction::ConcurrentMap_Leapfrog` tables when the load factor gets too high.

### 7.3.4 `cuckoohash_map`

`cuckoohash_map` [89] is a concurrent hash map that uses cuckoo hashing. This hash table has limited growth to a multiple of its original capacity. It uses fine-grained locking for synchronization. The paper also describes implementation that uses transactional memory, but this version is not yet available in this library. This map is included in the `libcuckoo` [98] header-only library.

# Chapter 8

# LSU3shell improvements

Almost all of the code that was tested is in a GitLab[33] repository in separate branches. When referring to branches and commits, it will be referring to this repository.

## 8.1 Userland allocators

When I first joined the project, the team already was not using the GNU C Library, but they were using tbbmalloc.

Most of the less popular userland memory allocators have been prone to being unstable in our tests. Hoard, TLSF, MCMalloc, scalloc, and StreamFlow all cause segmentation faults almost instantly. rpmalloc is able to run for a longer period of time, but ends up causing a segmentation fault most of the time also. A request for memory from bmalloc causes deadlock immediately. Both SFMalloc and SSMalloc run out of memory almost immediately and get killed by the OOM killer. The only stable ones for us were: glibc, jemalloc, litemalloc, LLAlloc, SuperMalloc, tbbmalloc, and tcmalloc. Results of a performance analysis of these userland allocators is shown on Figures 8.1, 8.2, 8.3, and 8.4.

Even though jemalloc is possible the most popular custom userland allocator right now, it the performance was really bad in our use-cases. Profiling has shown that the CPU is not properly utilized when using jemalloc, and a large amount of time is spent waiting on locks. This might be a problem with a specific CPU architecture, since we have not seen such abnormalities when measuring on the STAR cluster. When looking at voluntary context switches on dataset A, glibc makes 6,278,795,339 voluntary context switches, jemalloc makes 3,963,653,772 voluntary context switches, while all the other allocators make less than 40,000 voluntary context switches. This might explain the performance characteristic.

[33]https://gitlab.fit.cvut.cz/kocicma3/lsu3shell

LLAlloc has great performance characteristics, in some cases beating even tcmalloc, but it has almost quadruple the memory usage compared to other allocators. The memory usage does seem bounded though, so if the memory usage is not a problem, it might be an allocator worth considering.

Otherwise the measurements do not contain anything surprising—tcmalloc is the clear winner for our use-cases, and the three remaining userland allocators have similar performance characteristics.

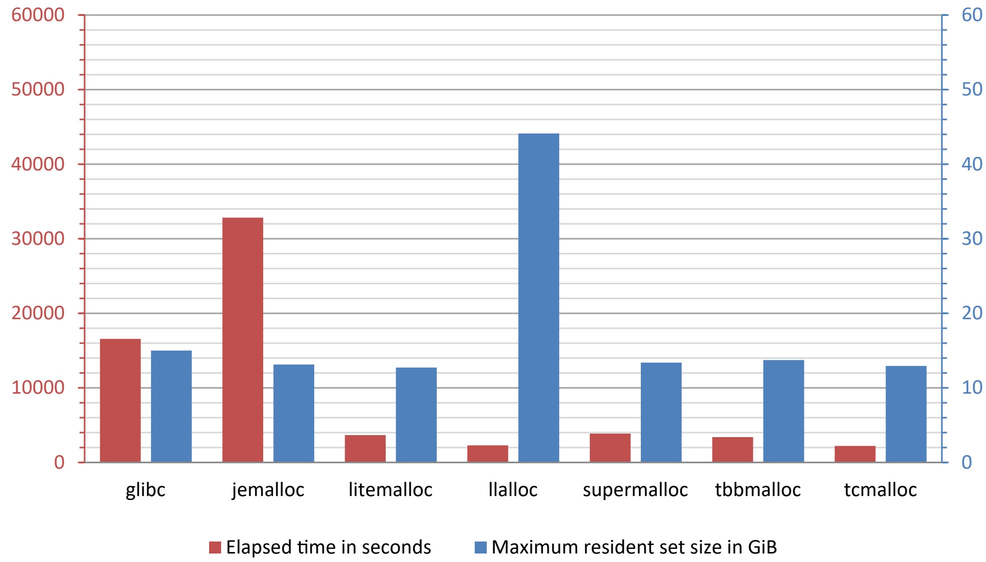



Figure 8.1: Comparison of time and memory utilization of different userland allocators on dataset A.

## 8.2 Memory pooling

From the analysis I picked two spots where pooling made most sense—they are in the hot path, and they allocate fixed-size memory blocks. When allocating arrays, the performance of memory pooling degrades significantly.

First is pooling `SU3xSU2::RME` instances in the `CRMECalculator` class (commit `99767b57`). Analysis has shown us that these allocations are never moved among threads, so each thread gets its own thread-local pool for these instances. Any attempt to use shared pools brought huge performance degradation, since these are contentious parts of the code, and even simplest lock-free synchronization can cause performance degradation.

Second was pooling instances of `CRMECalculator` (commit `d18200c7`) itself that are allocated in the `CTensorStructure::GetRMECalculator` and deallocated in classes `CTensorGroup_ada` and `CTensorGroup`. Same as with the `SU3xSU2::RME` instances, we observed that pair allocations and deallocations

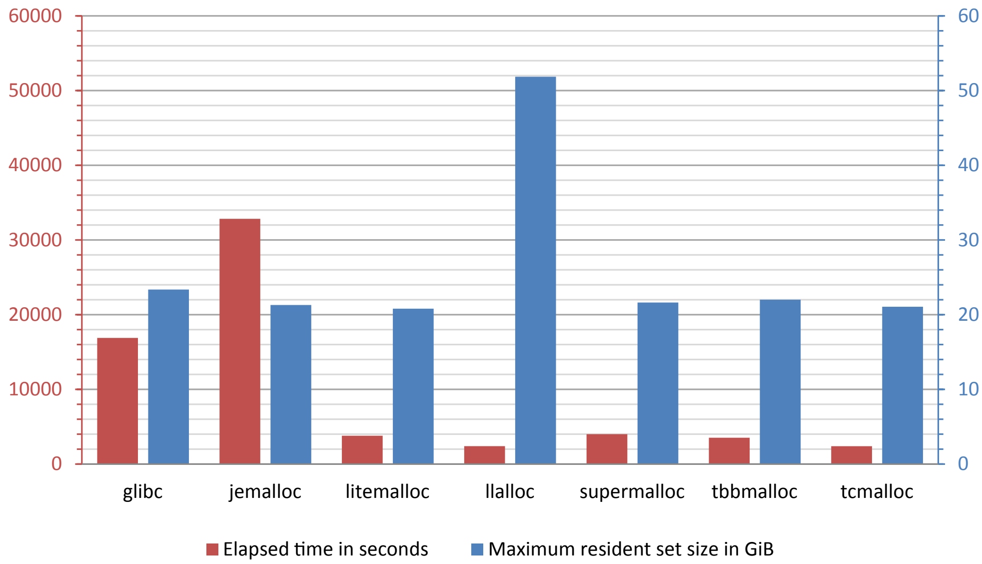



Figure 8.2: Comparison of time and memory utilization of different userland allocators on dataset B.

are always both requested from the same thread, so I used `thread_local` pool for these instances.

I did many more experiments with pooling, *e.g.*, branches `m_rme_pooling` and `fixed_m_rme_pooling`, but nothing else lead to significant performance gain.

Applying pooling in these two places led to reduction in memory allocation function calls by 13 %.

## 8.3 Small size optimization

`CTensorStructure::GetRMECalculator` is one of the most used functions in `LSU3shell`, and five small vectors are heavily used inside. Four of these vectors are as big as the number of shells occupied either in bra or ket state. Our measurements shown that this number is always under 16, and most of the time under 12, so these vectors seem as a great candidate for SSO. I used `boost::container::small_vector` with stack buffer size of 16. Even if there was more than 16 elements, the vector just switches to using heap memory. Experimenting with the stack buffer size might give us some interesting insights.

The results are shown in **??** and **??**. We see a significant reduction in run time, in some cases over 40 %. Maximum RSS slightly grew, but not enough to offset the performance benefits.

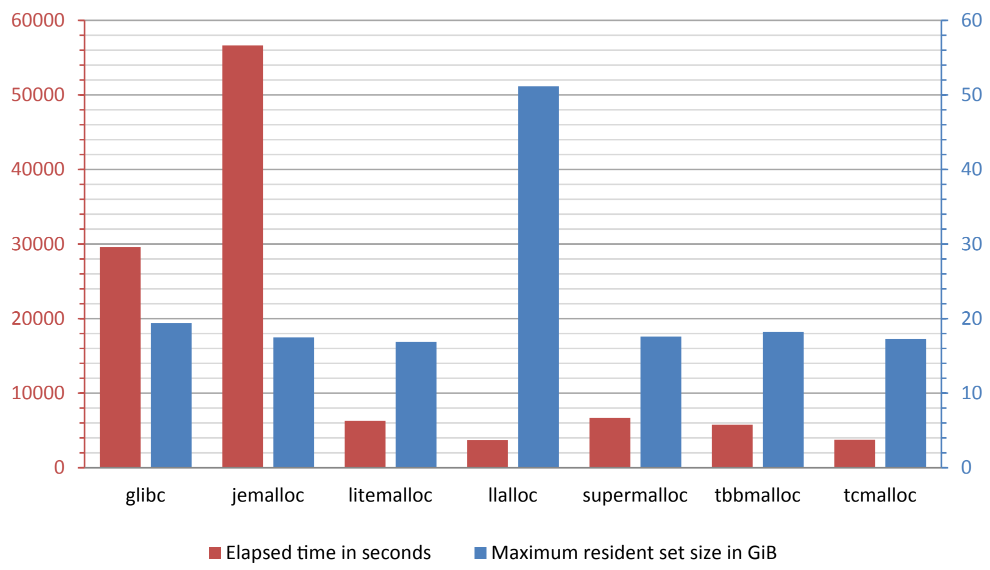



Figure 8.3: Comparison of time and memory utilization of different userland allocators on dataset C.

## 8.4 C++ STL memory allocators

Experimenting with both userland allocators and memory pooling had shown us, that any synchronization needed on memory allocation will lead to considerable performance degradation. There are two reasons why I decided not to used custom STL memory allocators. First, out of all of the implementations I have researched, none provide thread-local allocation buffers or pools, and we can not afford any additional synchronization. Second, even in places where we could gain performance by using a custom STL allocator, small size optimization can be used instead with far better performance.

## 8.5 Hash tables

### 8.5.1 Hash functions

I tried many different hash functions based on the SMHasher [83]: t1ha, MetroHash, SpookyHash, xxHash, and FarmHash. None of these gave us any significant speedup compared to `boost::hash_combine`. We can conclude that `boost::hash_combine` is fast enough and gives uniform enough distribution.

### 8.5.2 Other implementations

Usually the best performing implementations use atomic CAS operation to swap keys, thus limiting the keys to be at most 64 bit integers. The most

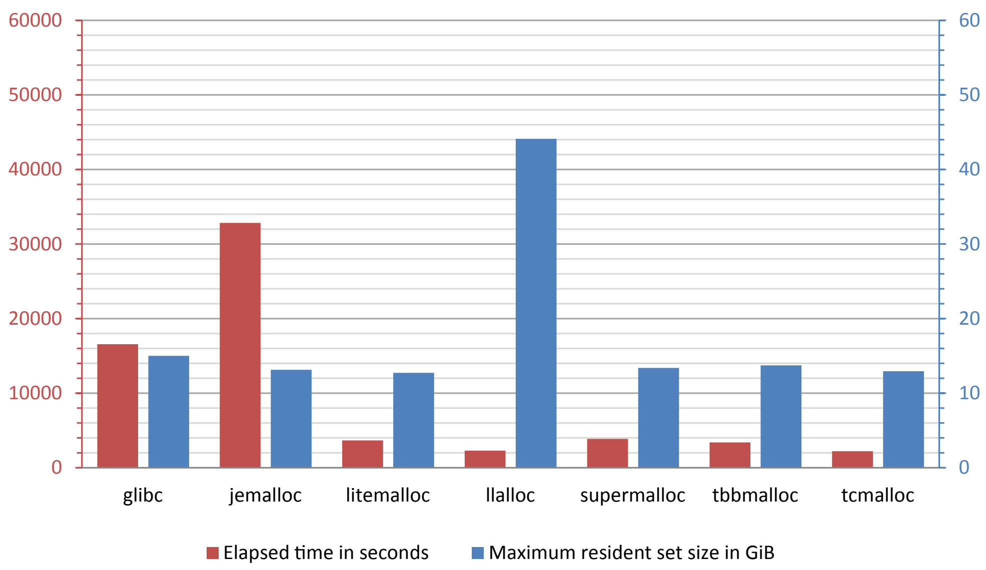



Figure 8.4: Comparison of time and memory utilization of different userland allocators on dataset D.

heavily used table in our code, the `CWig9lmLookUpTable` has a key that consists of 18 8 bit components that can not be made more compact in any way. Therefore these tables are not usable for our the table that is the biggest bottleneck.

Replacing `CWig9lmLookUpTable` with `cuckoohash_map` (implemented in branch `cwig9lm_libcuckoo`) was unstable, leading to consistent segmentation faults.

`folly::AtomicHashMap` is known to be very fast. Even though it only supports at most 64 bit keys, I have tried replacing `CWig9lmLookUpTable` with using only a part of the key (branch `cwig9lm_folly_atomic`), which of course leads to wrong results. I did this to try how fast `folly::AtomicHashMap` could be compared to our `HashFixed`. Surprisingly, `folly::AtomicHashMap` was only just as fast as `HashFixed` with no considerable difference in memory usage, showing that our implementation is very good despite its limitations. Tables without limitations on keys like `folly::ConcurrentHashMap` (branch `cwig9lm_folly_concurrent`) or `tbb::concurrent_unordered_map` (branch `cwig9lm_tbb_unordered`) all proven to be slower than our `HashFixed`, even though they have some additional features like record erasure and rehashing. The performance degradation did not justify these additional features for us.

## 8.6 Vectorization

Most of the loops in the hot path are very complicated, and thus not viable for vectorization. Those loops that are viable are vectorized automatically by the compiler. For example all mainstream C++ compilers (GCC, clang, Intel® icc, MSVC) were able to vectorize function `SU3xSU2::RME::rme2_x_su39lm_x_rme1` since 2009, as tested with the Compiler Explorer.

## 8.7 Results

Results for all memory optimizations and their interesting combinations are in Figures 8.5, 8.6, 8.7, 8.8. The bars (in this order) describe following scenarios:

1. The original code run with GNU C Library memory allocator.
2. The original code run with tbbmalloc memory allocator.
3. The original code run with tcmalloc memory allocator.
4. The code optimized with memory pools run with GNU C Library memory allocator.
5. The code optimized with small size optimization run with GNU C Library memory allocator.
6. The code optimized with memory pools and small size optimization run with GNU C Library memory allocator.
7. The code optimized with memory pools and small size optimization run with tcmalloc memory allocator (the final result).

Notice that the Y-axis on Figure 8.8 has logarithmic scale. As discussed before, glibc and jemalloc make abnormal amount of voluntary context switches.

We can see that with both memory pooling and small size optimizations applied, the gap between the better performing and worse performing userland allocators is becoming smaller. So even if, for some reason, the userland allocator can not be replaced, the performance degradation can be offset by using these techniques in user's code.

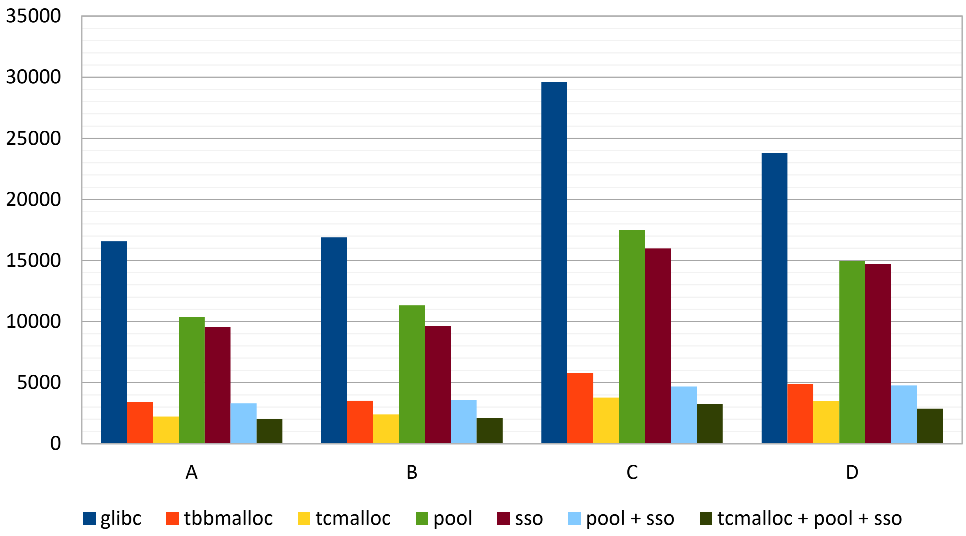



Figure 8.5: Time elapsed in seconds for all optimizations.

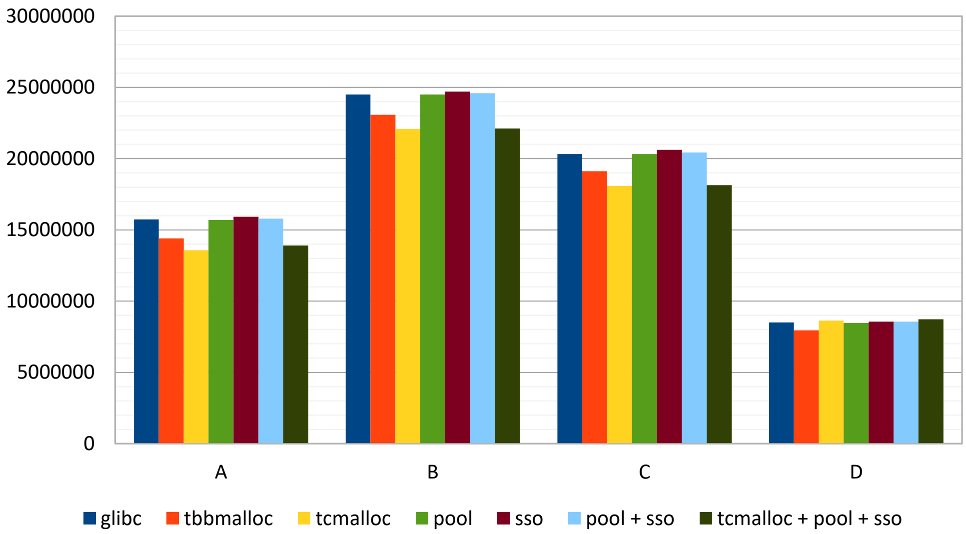



Figure 8.6: Maximum resident set size in GiB for all optimizations.

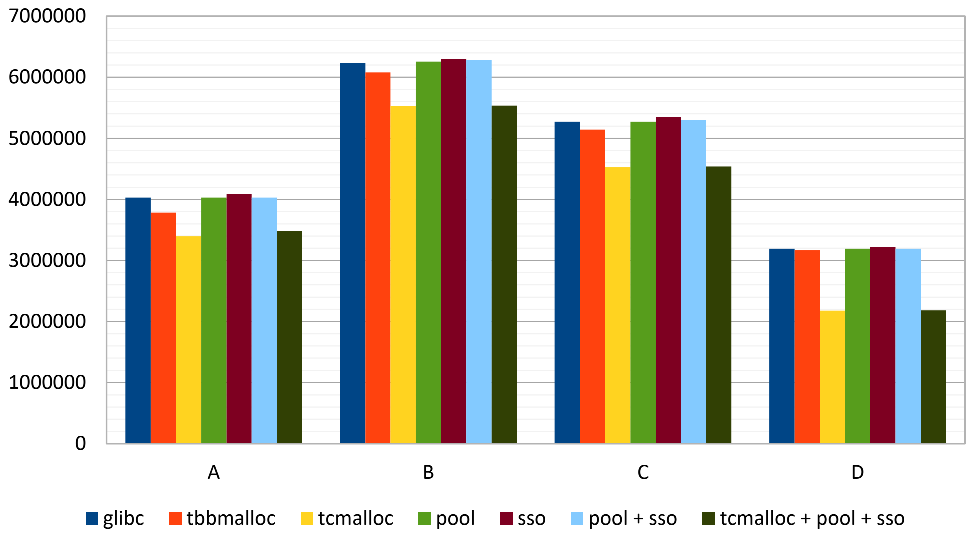



Figure 8.7: The number of minor page faults for all optimizations.

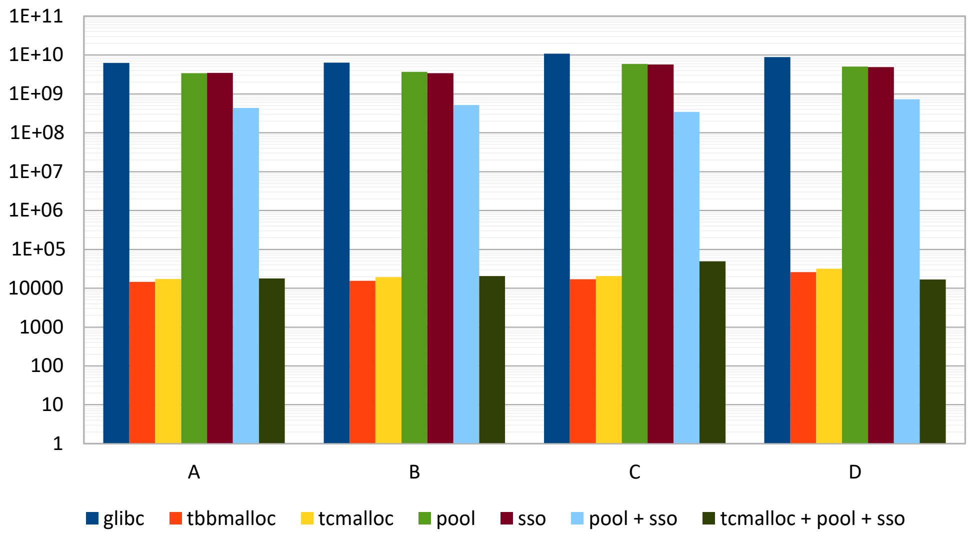



Figure 8.8: The number of voluntary context switches for all optimizations.

# Conclusion

The aim of this thesis has been to explore the problem of dynamic memory allocation, research the methods that can improve the performance of memory allocation-heavy code, analyze the performance of the `LSU3shell` program, and apply researched methods to the `LSU3shell` program.

Chapter 1 provided an overview of tools that can be used to analyze the performance of a program, and these tools were applied in chapter 2 to analyze the `LSU3shell` program.

Chapters 3 through 6 provided a detailed research into the problem of dynamic memory allocation and included an overview of existing solutions for some of the researched problems. This research was applied to the `LSU3shell` program in chapter 8 and the results were discussed.

I proposed several improvements to the `LSU3shell` program, implemented them, and measured the performance and memory consumption impact. I was able to reduce the run time by 41 % on average while slightly lowering the memory usage.

The optimizations introduced in this work will save our team up to 1.4 million core-hours of our total BlueWaters resource allocation, which I consider a success.

# Bibliography

[1] Evans, J. *jemalloc(3) User Manual*. Aug. 2018.

[2] Navrátil, P.; Quaglioni, S.; et al. Unified ab initio approaches to nuclear structure and reactions. *Physica Scripta*, volume 91, no. 5, 2015: p. 053002.

[3] Langr, D.; Dytrych, T.; et al. Efficient Parallel Generation of Many-Nucleon Basis for Large-Scale Ab Initio Nuclear Structure Calculations. In *International Conference on Parallel Processing and Applied Mathematics*, Springer, 2017, pp. 341–350.

[4] Edge, J. Perfcounters added to the mainline. July 2009, accessed January 9, 2019. Available from: `https://lwn.net/Articles/339361/`

[5] Gregg, B. Linux perf Examples. June 2018, accessed January 9, 2019. Available from: `www.brendangregg.com/perf.html`

[6] Berris, D. M.; Veitch, A.; et al. XRay: A function call tracing system. 2016.

[7] Hazelwood, K.; Kanev, S.; et al. Profiling a Warehouse-Scale Computer. 2015.

[8] Wilson, P. R.; Johnstone, M. S.; et al. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, London, United Kingdom: Springer-Verlag, 1995, ISBN 3-540-60368-9, pp. 1–116.

[9] Larson, P.-Å.; Krishnan, M. Memory allocation for long-running server applications. In *ACM SIGPLAN Notices*, volume 34, ACM, 1998, pp. 176–185.

[10] Kanev, S.; Xi, S. L.; et al. Mallacc: Accelerating Memory Allocation. *ACM SIGARCH Computer Architecture News*, volume 45, no. 1, 2017: pp. 33–45.

[11] Free Software Foundation, Inc. *objdump(1) GNU Development Tools*. July 2018.

[12] Aigner, M.; Kirsch, C. M.; et al. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In *ACM SIGPLAN Notices*, volume 50, ACM, 2015, pp. 451–469.

[13] Kuszmaul, B. C. SuperMalloc: A Super Fast Multithreaded Malloc for 64-bit Machines. *SIGPLAN Not.*, volume 50, no. 11, June 2015: pp. 41–55, ISSN 0362-1340, doi:10.1145/2887746.2754178.

[14] Herlihy, M.; Shavit, N. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[15] Free Software Foundation, Inc. Memory Concepts (The GNU C Library). 2018, accessed November 25, 2018. Available from: `https://www.gnu.org/software/libc/manual/html_node/Memory-Concepts.html`

[16] Free Software Foundation, Inc. Locking Pages (The GNU C Library). 2018, accessed November 26, 2018. Available from: `https://www.gnu.org/software/libc/manual/html_node/Locking-Pages.html`

[17] Institute of Electrical and Electronics Engineers. *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7*. Jan. 2018.

[18] Free Software Foundation, Inc. Locked Memory Details (The GNU C Library). 2018, accessed November 26, 2018. Available from: `https://www.gnu.org/software/libc/manual/html_node/Locked-Memory-Details.html`

[19] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2018.

[20] ARM Holdings. *ARM® Architecture Reference Manual: ARMv8, for ARMv8-A architecture profile*. 2017.

[21] International Business Machines Corporation. *Power ISA™ Version 3.0 B*. 2017.

[22] Oracle. *UltraSPARC Architecture 2007*. 2010.

[23] Intel Corporation. *Intel® Itanium® Architecture Software Developer's Manual*. 2010.

[24] Arcangeli, A. 20 years of Linux Virtual Memory: from simple server workloads to cloud virtualization. Presented at the FOSDEM conference in Brussels, Belgium on February 4, 2017. Available from: `https://archive.fosdem.org/2017/schedule/event/iaas_20yealin/attachments/slides/1498/export/events/attachments/iaas_20yealin/slides/1498/VM.pdf`

[25] Munson, E.; Gorman, M.; et al. libhugetlbfs/libhugetlbfs. Accessed December 2, 2018. Available from: `https://github.com/libhugetlbfs/libhugetlbfs`

[26] Rapoport, M. Transparent Hugepage Support. *Linux Kernel Documentation*, 2018.

[27] Nikitin, A. Transparent Hugepages: measuring the performance impact - The mole is digging. 2017, accessed December 2, 2018. Available from: `https://alexandrnikitin.github.io/blog/transparent-hugepages-measuring-the-performance-impact/`

[28] Institute of Electrical and Electronics Engineers. *IEEE Standard for IEEE Information Technology - Portable Operating System Interface (POSIX(R))*. Jan. 2018.

[29] *madvise(2) Linux Programmer's Manual*. Sept. 2017.

[30] *International Standard ISO/IEC 9899:2018(E) – Information Technology — Programming Languages — C*. Geneva, Switzerland: International Organization for Standardization (ISO), July 2018.

[31] Kerrisk, M. *The Linux Programming Interface: A Linux and UNIX® System Programming Handbook*. No Starch Press, 2010.

[32] Berger, E. D.; McKinley, K. S.; et al. Hoard: A scalable memory allocator for multithreaded applications. In *ACM SIGARCH Computer Architecture News*, volume 28, ACM, 2000, pp. 117–128.

[33] Tadman, M. Fast-t: A new hierarchical dynamic storage allocation technique. *Master's thesis, UC Irvine*, 1978.

[34] Free Software Foundation, Inc. The GNU C Library. 2018, accessed November 13, 2018. Available from: `https://www.gnu.org/software/libc/libc.html`

[35] Gloger, W. Wolfram Gloger's malloc homepage. 2006, accessed February 16, 2018. Available from: `http://www.malloc.de/en/`

[36] Baldassin, A.; Borin, E.; et al. Performance Implications of Dynamic Memory Allocators on Transactional Memory Systems. *SIGPLAN Not.*, volume 50, no. 8, Jan. 2015: pp. 87–96, ISSN 0362-1340, doi:10.1145/2858788.2688504.

[37] Lea, D. A Memory Allocator. 2000, accessed February 16, 2018. Available from: `http://g.oswego.edu/dl/html/malloc.html`

[38] *mallopt(3) Linux Programmer's Manual*. Sept. 2017.

[39] Knuth, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. 1973, ISBN 9780321635747.

[40] Free Software Foundation, Inc. Tunables (The GNU C Library). 2018, accessed November 19, 2018. Available from: `https://www.gnu.org/software/libc/manual/html_node/Tunables.html`

[41] Free Software Foundation, Inc. Memory Allocation Tunables (The GNU C Library). 2018, accessed November 19, 2018. Available from: `https://www.gnu.org/software/libc/manual/html_node/Memory-Allocation-Tunables.html`

[42] Ghemawat, S.; Menage, P. TCMalloc : Thread-Caching Malloc. 2007, accessed November 5, 2018. Available from: `https://gperftools.github.io/gperftools/tcmalloc.html`

[43] Lee, S.; Johnson, T.; et al. Feedback directed optimization of TCMalloc. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, ACM, 2014, p. 3.

[44] Evans, J. A Scalable Concurrent `malloc(3)` Implementation for FreeBSD. 2006.

[45] Kamp, P.-H. Malloc (3) revisited. In *USENIX Annual Technical Conference*, 1998, p. 45.

[46] Evans, J.; Wang, Q.; et al. jemalloc/jemalloc. Accessed November 2, 2018. Available from: `https://github.com/jemalloc/jemalloc`

[47] Argyroudis, P.; Karamitas, C. Exploiting the jemalloc Memory Allocator: Owning Firefox's Heap. 2012.

[48] Kukanov, A.; Voss, M. J. The Foundations for Scalable Multi-Core Software in Intel® Threading Building Blocks. In *Intel® Technology Journal*, Volume 11, Issue 04, 2007, pp. 309–322.

[49] Intel Corporation. Threading Building Blocks. 2018. Available from: `https://www.threadingbuildingblocks.org`

[50] Hudson, R. L.; Saha, B.; et al. McRT-Malloc: A Scalable Transactional Memory Allocator. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, New York, NY, USA: ACM, 2006, ISBN 1-59593-221-6, pp. 74–83, doi:10.1145/1133956.1133967.

[51] Reinders, J. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O'Reilly, first edition, 2007, ISBN 0596514808.

[52] Březina, P. Talloc - a hierarchical memory allocator. *Bachelor's thesis, Masaryk University*, 2012.

[53] Masmano, M.; Ripoll, I.; et al. TLSF: A new dynamic memory allocator for real-time systems. 2004.

[54] Schneider, S.; Antonopoulos, C. D.; et al. Scalable locality-conscious multithreaded memory allocation. 2006.

[55] Jansson, M. rampantpixels/rpmalloc: Public domain cross platform lock free thread caching 32-byte aligned memory allocator implemented in C. Accessed January 6, 2019. Available from: `https://github.com/rampantpixels/rpmalloc`

[56] Manghwani, R.; He, T. Scalable memory allocation. 2011.

[57] Umayabara, A.; Yamana, H. MCMalloc: A scalable memory allocator for multithreaded applications on a many-core shared-memory machine. In *Big Data (Big Data), 2017 IEEE International Conference on*, IEEE, 2017, pp. 4846–4848.

[58] Liu, R.; Chen, H. SSMalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the Asia-Pacific Workshop on Systems*, ACM, 2012, p. 15.

[59] Seo, S.; Kim, J.; et al. SFMalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, IEEE, 2011, pp. 253–263.

[60] *International Standard ISO/IEC 14882:2017(E) – Programming Languages — C++*. Geneva, Switzerland: International Organization for Standardization (ISO), Dec. 2017.

[61] Stepanov, A.; Lee, M. *The Standard Template Library.* HP Laboratories, 1995.

[62] Halpern, P. N1850: Towards a Better Allocator Model. 2005. Available from: `https://wg21.link/N1850`

[63] *International Standard ISO/IEC 14882:2011(E) – Information Technology — Programming Languages — C++*. Geneva, Switzerland: International Organization for Standardization (ISO), Sept. 2011.

[64] N4791: Working Draft, Standard for Programming Language C++. Dec. 2018. Available from: https://wg21.link/N4791

[65] Halpern, P. CppCon 2017: Pablo Halpern "Allocators: The Good Parts". Available from: https://www.youtube.com/watch?v=v3dz-AKOVL8

[66] Meredith, A. Allocators in C++11. Presented at the C++Now conference in Aspen, Colorado on May 16, 2013.

[67] Gaztañaga, I. N2045: Improving STL Allocators. 2006. Available from: https://wg21.link/N2045

[68] Hinnant, H. N1953: Upgrading the Interface of Allocators using API Versioning. 2006. Available from: https://wg21.link/N1953

[69] Meredith, A. CppCon 2017: Alisdair Meredith "An allocator model for std2". Available from: https://www.youtube.com/watch?v=oCi_QZ6K_qk

[70] Bloomberg L.P. bloomberg/bde. Accessed December 16, 2018. Available from: https://github.com/bloomberg/bde

[71] Halpern, P. N3916: Polymorphic Memory Resources, 2nd revision. 2014. Available from: https://wg21.link/N3916

[72] Boyall, M. N3575: Additional Standard allocation schemes. 2013. Available from: https://wg21.link/N3575

[73] Diduck, L. N2486: Alternative Allocators and Standard Containers. 2007. Available from: https://wg21.link/N2486

[74] Lakos, J.; Mendelsohn, J.; et al. P0089: On Quantifying Memory-Allocation Strategies, 2nd revision. 2016. Available from: https://wg21.link/P0089

[75] Bleaney, G. P0213: Validation of Memory-Allocation Benchmarks. 2016. Available from: https://wg21.link/P0213

[76] Moerbeek, O. A new malloc(3) for OpenBSD. Presented at the European BSD Conference in Cambridge, England on September 19, 2009.

[77] Electronic Arts Inc. electronicarts/EASTL. Accessed December 28, 2018. Available from: https://github.com/electronicarts/EASTL

[78] Pedriana, P. N2271: EASTL – Electronic Arts Standard Template Library. 2007. Available from: https://wg21.link/N2271

[79] Jula, A.; Rauchwerger, L. Two Memory Allocators That Use Hints to Improve Locality. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, New York, NY, USA: ACM, 2009, ISBN 978-1-60558-347-1, pp. 109–118, doi:10.1145/1542431.1542447.

[80] Ormrod, N. CppCon 2016: Nicholas Ormrod "The strange details of std::string at Facebook". Available from: `https://www.youtube.com/watch?v=kPR8h4-qZdk`

[81] Gadeschi, G. B. P0843: static_vector, 2nd revision. 2018. Available from: `https://wg21.link/P0843`

[82] Appleby, A. aappleby/smhasher: Automatically exported from code.google.com/p/smhasher. Accessed January 5, 2019. Available from: `https://github.com/aappleby/smhasher`

[83] Urban, R. rurban/smhasher: Improved fork of https://code.google.com/p/smhasher/. Accessed January 6, 2019. Available from: `https://github.com/rurban/smhasher`

[84] Cormen, T. H.; Leiserson, C. E.; et al. *Introduction to algorithms*. MIT press, 2009.

[85] Pagh, R.; Rodler, F. F. Cuckoo hashing. *Journal of Algorithms*, volume 51, no. 2, 2004: pp. 122–144.

[86] Herlihy, M.; Shavit, N.; et al. Hopscotch hashing. In *International Symposium on Distributed Computing*, Springer, 2008, pp. 350–364.

[87] Fan, B.; Andersen, D. G.; et al. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI*, volume 13, 2013, pp. 371–384.

[88] Zhou, D.; Fan, B.; et al. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, ACM, 2013, pp. 97–108.

[89] Li, X.; Andersen, D. G.; et al. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, ACM, 2014, p. 27.

[90] Shalev, O.; Shavit, N. Split-Ordered Lists: Lock-free Extensible Hash Tables. *Journal of the ACM (JACM)*, volume 53, no. 3, 2006: pp. 379–405.

[91] Malakhov, A. A. Per-bucket concurrent rehashing algorithms. *arXiv preprint arXiv:1509.02235*, 2015.

[92] Ahrens, S.; DeLong, J. *folly/AtomicHashmap.h.* Nov. 2016. Available from: https://github.com/facebook/folly/blob/master/folly/docs/AtomicHashMap.md

[93] Ahrens, S. Massively Parallel Datastructures. Presented at the Facebook C++ Conference in Menlo Park, California on June 2, 2012. Available from: https://allfacebook.de/wp-content/uploads/2012/06/Massively_Parallel_Datastructures.pdf

[94] Preshing, J. preshing/junction: Concurrent data structures in C++. Accessed January 6, 2019. Available from: https://github.com/preshing/junction

[95] Hart, T. E. Comparative performance of memory reclamation strategies for lock-free and concurrently-readable data structures. *Master's thesis, University of Toronto*, 2005.

[96] Click, C. Advanced Topics in Programming Languages: A Lock-Free Hash Table. Presented at Google Tech Talks on March 28, 2007. Available from: https://www.youtube.com/watch?v=HJ-719EGIts

[97] Preshing, J. New Concurrent Hash Maps for C++. 2016, accessed January 6, 2019. Available from: https://preshing.com/20160201/new-concurrent-hash-maps-for-cpp/

[98] Goyal, M.; Scott, J. S.; et al. efficient/libcuckoo: A high-performance, concurrent hash table. Accessed January 6, 2019. Available from: https://github.com/efficient/libcuckoo

Appendix A

# Acronyms

**API** Application Programming Interface

**ASLR** Address space layout randomization

**CAS** Compare-and-swap

**CPU** Central processing unit

**GNU** GNU is a recursive acronym for “GNU’s Not Unix!”

**GPU** Graphics processing unit

**HPC** High Performance Computing

**HTM** Hardware Transactional Memory

**MPI** Message Passing Interface

**NUMA** Non-uniform memory

**POSIX** Portable Operating System Interface [for Unix]

**SMP** Symmetric Multiprocessing

**SSO** Small Size Optimization

**STL** Standard Template Library

**TLB** Translation lookaside buffer

**TLS** Thread-local storage

Appendix B

# Contents of enclosed CD

```
readme.txt ............ the file with CD contents description
src ........................ the directory of source codes
  thesis ........ the directory of LaTeX source codes of the thesis
text ........................ the thesis text directory
  thesis.pdf ................ the thesis text in PDF format
```