Scalloc Artifact Overview

Martin Aigner Christoph M. Kirsch Michael Lippautz Ana Sokolova
University of Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

Contents

Synopsis

The scalloc artifact allows evaluating memory allocators using a set of predefined workloads and settings, including all experiments presented in the paper. The artifact is designed to be modular, i.e., benchmarks and allocators can be added independently from each other. The artifact, containing only open-source software, is bootstrapped from a single self-hosting repository. Dependencies and software that is not open-source are retrieved as needed (and as the license allows). Table 1 illustrates the software sources needed to evaluate scalloc. All allocators are retrieved and built using ACDC, an open-source benchmark that may be configured to emulate explicit single- and multi-threaded memory allocation, sharing, access, and deallocation behavior to expose virtually any relevant allocator performance differences.

Throughout this document we refer to sections in the main paper as *MP Section <number>*.

ARTIFACT SOURCE	
Self-hosting Scalloc Artifact	`https://github.com/cksystemsgroup/scalloc-artifact`
STAND-ALONE REPOSITORIES	
scalloc	`https://github.com/cksystemsgroup/scalloc`
ACDC	`https://github.com/cksystemsgroup/ACDC`
Hoard benchmarks	`https://github.com/emeryberger/Hoard`

Table 1: Artifact sources

1. Getting Started

To avoid the trouble of setting up an environment, the artifact ships with a virtual machine (VM) image in which all benchmarks are already set up appropriately. Hence, the VM image can also be used offline. This section guides through the process of configuring the provided virtual machine, provides an overview of the artifacts directory and file structure, and finally gives an example of how to run a single benchmark. Please note that the VM image serves as playground for the artifact. For reproducing performance results we recommend setting up the artifact natively without virtualization.

1.1 Virtual Machine Image

The provided virtual machine image, `Scalloc Artifact.vdi`, has been created using the VirtualBox[1] hypervisor. However, any software capable of importing VirtualBox images may potentially be used.

The operating system in the VM image is set up as Ubuntu 14.04 LTS Desktop including latest updates (point releases). The system is set up with a Gnome UI and a running ssh daemon on port 22. The Gnome UI is only provided for convenience. All benchmarks can be executed through command line scripts. The following username and password is used throughout working on the VM (the user has superuser permissions):

user:	artifact
password:	artifact

VirtualBox Settings

The VirtualBox settings file for the VM image, `Scalloc Artifact.vbox`, is configured as illustrated in Table 2.

Physical address space extension (PAE)	enabled
Hardware virtualization (VT-x/AMD-V)	enabled
Nested paging	enabled
3D graphics acceleration	disabled
2D graphics acceleration	disabled
Available cores	16
RAM size	60GB

Table 2: VirtualBox configuration for the provided VM image

Running the VM and in particular executing the benchmarks may result in different observations if the underlying hardware cannot provide sufficient resources (cores, RAM). For running VirtualBox in headless mode (without a GUI) consult the user manual[2] on changing presets using the `VBoxManage` command. Note that the artifact has actually been created in headless mode, hence the disabled graphics accelerations.

1.2 Folder Structure

The artifact is installed into the artifact user's `Desktop` folder, i.e., it can be found in `~/Desktop/scalloc-artifact`. We refer to this directory as `$ARTIFACT_HOME` throughout this document. Note that some of the described folders and files may only be present after running a benchmark.

`env.sh`

This script contains basic parameters for all benchmarks, e.g., how many repetitions are recorded for a single data point and the range of threads to be benchmarked. The script ships with two predefined configurations: `fast` and `paper`, which can be used for trying out benchmarks and reproducing results from the paper,

[1] Available online at `https://www.virtualbox.org/`.

[2] Available online at `https://www.virtualbox.org/manual/UserManual.html`.

respectively. The following snippet shows how the script needs to be sourced with the corresponding configuration before executing a benchmark

```
$ source env.sh fast  # may require typing the password
$ # or, for the paper configuration
$ source env.sh paper # may require typing the password
```

run/
: This directory contains the scripts that are used to execute benchmarks. After sourcing the environment, benchmarks are executed from $ARTIFACT_HOME following the pattern

```
$ ./run/<benchmark to run> <additional paramters>
```

tools/
: This directory contains the scripts used to bootstrap the artifact directory.

tools/make_deps.sh
: The script is used to install operating system dependencies and software that is needed for creating the artifact and building included software.

tools/create_env.sh
: The script is used to bootstrap the artifact. The script downloads all necessary sources for all benchmarks and allocators, configures them, and finally builds them.

tools/delete_env.sh
: The script cleans the artifact, i.e., it removes all generated output including compiled benchmarks and data files.

benchmarks/
: This directory contains sources and the resulting compiled binaries for all benchmarks. Note that some directories are only present after creating the environment. For example, ACDC is retrieved from its online repository.

allocators/
: We use ACDC to retrieve and build allocators. After creating the artifact environment, this directory is merely a symbolic link into ACDC's allocator directory.

data/
: After executing a benchmark the resulting data files can be found in this directory. Note that benchmarks create subdirectories, hence do not overwrite their results.

plot/
: After executing a benchmark the resuting data can be plotted using the gnuplot scripts in this directory. Note that the included files are only skeletons that potentially require refinement.

1.3 Example: Running a Single Experiment

This example demonstrates running a single experiment using a shell script shipped in the run/ directory. The following steps apply to any experiment that can be found in the run/ directory. Note that even the fast artifact configuration includes all allocators listed in the ALLOCATORS variable in env.sh. For testing purposes, certain allocators can be excluded by editing env.sh and removing them from ALLOCATORS.

```
$ # from within $ARTIFACT_HOME
$ source env.sh fast          # environment for a quick demo run
$ ./run/acdc-prod-cons.sh     # execute ACDC's producer/consumer experiment
```

The data is written to the corresponding subdirectory in data, i.e., data/ACDC/prod-cons/ for this experiment. The format of the data files, for each of the corresponding experiments, is described in Section 2.

2. Reproducing Results

The following section provides descriptions on how to reproduce the results illustrated in the main paper.

2.1 Hardware and Software Setup

Hardware

Our data can be reproduced on the machine described in detail in *MP Section 4*. Note that architectural differences might affect the results. For closest reproduction we recommend running the artifact on SMP non-NUMA hardware because scalloc currently does not address the performance characteristics of NUMA machines. Differences in other architectural aspects may change absolute values but should preserve the relative performance differences among different allocators.

Software

The artifact and all experiments have been run on Ubuntu 14.04 LTS. The setup as pre-installed in the VM image can be obtained in the following way.

```
$ git clone -b 1.0.0 https://github.com/cksystemsgroup/scalloc-artifact.git
$ cd scalloc-artifact
$ tools/make_deps.sh  # may require typing the password
$ tools/create_env.sh        # may require typing the password
```

Before running an experiment it is important to adopt settings for certain Linux kernel features as they influence benchmarking.

- Allow overcommiting virtual memory as otherwise extensive use of virtual memory may be prohibited.
- Disable transparent hugepages as otherwise madvise and related syscalls may have no effect as the kernel is allowed to use hugepages to hold regular pages.
- Disable page defragmentation for the same reason as disabling transparent hugepages.

Note that env.sh takes care of those settings.

2.2 ACDC Benchmarks

All ACDC-based experiments can be executed with the corresponding script in the run/ directory. The script names are prefixed with acdc-. The data is then written to data/ACDC/[experiment name]/. ACDC writes one file per allocator and per metric named [allocator]-[metric].dat for the metrics: access (total memory access time where applicable), alloc (total time spent in malloc), free (total time spent in free), combined (total time spent in malloc and free), and memcons (average memory consumption). Temporal data is presented in CPU cycles and spatial data is presented in kilobytes.

The files generating and representing the data presented in the paper are as follows:

Producer-Consumer Workload *MP Section 4.3*

- run/acdc-prod-cons.sh
- data/ACDC/prod-cons/[allocator]-combined.dat
- data/ACDC/prod-cons/[allocator]-memcons.dat

Robustness for Varying Object Sizes *MP Section 4.5*

- run/acdc-object-size.sh

- data/ACDC/object-size/[allocator]-combined.dat
- data/ACDC/objext-size/[allocator]-memcons.dat

Mutator Performance: Locality *MP Section 4.6*

- run/acdc-locality.sh
- data/ACDC/locality/[allocator]-access.dat

Design Decisions: Virtual Spans *MP Section 7.7.1*

- run/acdc-virtual-spans-evaluation.sh
- data/ACDC/virtual-spans-evaluation/[allocator]-combined.dat
- data/ACDC/virtual-spans-evaluation/[allocator]-memcons.dat

Data File Layout

Listing 1 illustrates an example of the data file that contains the combined time spent in malloc and free for the llalloc allocator in the producer-consumer experiment: `data/ACDC/prod-cons/llalloc-combined.dat`. The lines starting with # contain information about the experiment, e.g., the options used to run ACDC. The parameters are explained in detail in the corresponding publication on ACDC. Each data file contains three tab-separated columns. The first column shows the factor values that are changed in the experiment, i.e., usually the values on the x-axis. The second column is the arithmetic mean of the measured values. The third column lists the sample standard deviation. Note that the data file can be used directly as source file for gnuplot figures.

Listing 1: Data file example for ACDC

```
#Created at: Tue Jun 2 14:29:49 CEST 2015 on big-iron8
#Average on 2 runs. ACDC Options: -a -s 4 -S 9 -d 30 -l 1 -L 5 -t 1000000 ...
#
#x(-n)   average        stddev

1          81691280.0   1508552.9
4         148391160.7   1415310.2
8         166494495.9   2656742.8
12        176680835.8   5371155.8
```

2.3 Non-ACDC Benchmarks

Similar to ACDC-based experiments, all other experiments can also be executed using the corresponding scripts in the `run/` directory. For a description of the workloads and parameters see Section 4 in the main paper.

The scripts can all be executed as a single or averaged run. All experiments are configured as described in the main paper. Temporal data is reported as specified in the main paper which usually amounts to total execution time in milliseconds or operations per second. Spatial data is reported as averaged memory consumption (resident set size) in kilobytes. Single runs do not record any data. Averaged runs record averaged and raw (the single run) data. The files generating and representing the data presented in the paper are as follows:

Threadtest *MP Section 4.2*

- run/threadtest-single.sh
- run/threadtest-averaged.sh
- run/threadtest-averaged-for-all-allocators.sh
- data/threadtest/[allocator].dat
- data/threadtest/[allocator]-raw.dat

Shbench *MP Section 4.2*

- `run/shbench-single.sh`
- `run/shbench-averaged.sh`
- `run/shbench-averaged-for-all-allocators.sh`
- `data/shbench/[allocator].dat`
- `data/shbench/[allocator]-raw.dat`

Larson *MP Section 4.2.1*

- `run/larson-single.sh`
- `run/larson-averaged.sh`
- `run/larson-averaged-for-all-allocators.sh`
- `data/larson/[allocator].dat`
- `data/larson/[allocator]-raw.dat`

Robustness against False Sharing *MP Section 4.4*

- `run/active-false-single.sh`
- `run/active-false-averaged.sh`
- `run/active-false-averaged-for-all-allocators.sh`
- `data/active-false/[allocator].dat`
- `data/active-false/[allocator]-raw.dat`
- `run/passive-false-single.sh`
- `run/passive-false-averaged.sh`
- `run/passive-false-averaged-for-all-allocators.sh`
- `data/passive-false/[allocator].dat`
- `data/passive-false/[allocator]-raw.dat`

Design Decisions: Scalable Backend *MP Section 7.7.2*

- `run/scalloc-evaluate-span-pool.sh`
- `data/span-pool-evaluation/[allocator].dat`

Design Decisions: Constant-time Frontend *MP Section 7.7.3*

- `run/scalloc-evaluate-frontend-cleanup.sh`
- `data/scalloc-frontend-evaluation/[allocator].dat`

Example: Threadtest

The following description shows the setup of the threadtest benchmark. Similarly, all other experiments can be run by executing the corresponding script. Note that all scripts executed without any parameters will inform the user about how to run the experiments.
A single run of threadtest can be executed using

```
$ source env.sh paper
$ # run for 1 threads in paper configuration
$ ./run/threadtest-single.sh \
      $ARTIFACT_HOME/allocators/libscalloc.so 1 "10000␣100000␣0␣8"
```

For a single allocator, an averaged run can be executed using

```
$ source env.sh paper
$ # run for a range of threads in paper configuration for one allocator
$ ./run/threadtest-averaged.sh \
     scalloc $ARTIFACT_HOME/allocators/libscalloc.so
```

Finally, the whole experiment for all allocators in the paper configuration can be executed using

```
$ source env.sh paper
$ ./run/threadtest-averaged-for-all-allocators.sh
```

Data File Layout

Listing 2 illustrates an example of the data file for an allocator for the averaged threadtest run. The lines starting with # contain information about the experiment, e.g., the options to run threadtest. The parameters are explained in detail in the corresponding publication. Each data file contains five comma-separated columns. The first column represents the number of threads. The second and third column provide the average and standard deviation of the measured value (e.g. execution time), respectively. The fourth and five column provide the average and standard deviation of the measured memory consumption (RSS), respectively. Note that the data file can be used directly as source file for gnuplot figures.

Listing 2: Data file example for threadtest

```
# ./run/threadtest-single.sh <parameters>
# recorded at: Mit Jun  3 13:00:29 CEST 2015
# repetitions: 2
# threads, avg(execution time), corrected sample stddev(execution time), ...
1,153.426106,0.630731,9925.000000,42.426407
4,41.174034,0.700698,9750.000000,118.793939
8,26.993026,0.655759,9526.000000,94.752309
12,23.586545,0.206058,7555.000000,11.313708
```

2.4 SPEC CPU2006 483.xalancbmk

Due to licensing it is not possible to ship the SPEC CPU2006 483.xalancbmk benchmark. For completeness and reproducibility we provide the corresponding script to run the experiment in `run/spec.sh`. Note that SPEC provides its own settings on repetitions and machine configurations which need to be configured before running the experiment.

2.5 Plotting Data

Measurement data can be plotted using gnuplot [3] starting from version 4.6. After running a benchmark the corresponding data can be plotted using the following command. The resulting plots can be found in `out/`.

```
$ # after running a benchmark,  e.g. threadtest:
$ gnuplot plot/threadtest.gp
```

[3] `http://www.gnuplot.info/`

3. Scalloc Source Code

The following section provides an overview of scalloc's source code and pointers to the concepts of the main paper where appropriate. The source code can be found in `$ARTIFACT_HOME/allocators/scalloc`.

`src/arena.h` The arena to carve off virtual spans.	*MP Section 2.2*
`src/atomic_value.h` Tagged atomic values for 64-bit platforms.	*MP Section 2.5*
`src/core.h`, `src/core_id.h` Scalloc's main algorithm.	*MP Section 2.5*
`src/deque.h` A lock-based double-ended queue.	*MP Section 2.5*
`src/free_list.h` The incremental free-list implementation that is contained in real-spans.	
`src/globals.h` Global constants, e.g., size-class boundaries and arena sizes.	
`src/glue.h`, `src/glue.cc` Functions and utilities needed for glueing together all algorithms in an allocator.	
`src/lab.h` Thread-local allocation buffers (TLABs) and core-local implementation buffers (CLABs).	*MP Section 1*
`src/large-objects.h` The fallback allocator used for huge objects (the file name is legacy naming and will be changed to huge objects).	*MP Section 1*
`src/lock.h` A lock that is based on busy-waiting.	
`src/log.h` A non-intrusive logging implementation that can be used to trace allocation and deallocation requests.	
`src/size_classes.h`, `src/size_classes_raw.h` Size classes (bins) and utility functions.	Section 2.1
`src/span.h` The span abstraction.	throughout *MP Section 2*
`src/span_pool.h` The span-pool backend.	*MP Section 2.3*
`src/stack.h` A lock-free Treiber stack implementation.	throughout *MP Section 2*
`src/utils.h`, `src/platform/*` Utility functions and platform-specific overrides.	

4. Revision History

Revision	Date	Author(s)	Description
1.0	06/08/2015	scalloc authors	Initial version.
1.1	07/27/2015	scalloc authors	Added experiments from revised paper version.
1.2	08/11/2015	scalloc authors	Revised public version.