Fall'21. CS-263 PROJECT REPORT
Inspection of Julia GC for multithreaded programs

By: Deept Mahendiratta
Nazerke Turtayeva

Contents

Introduction to Julia Language

Julia is a high level, general purpose, dynamic programming language well-perceived for data analysis and computational research. It strives to employ the best of all worlds and aims for highly optimizable performance as C, for effortless and intuitive code as Python, for math friendly syntax as Matlab, for powerful string processing as Perl and many other acknowledged paradigms of different languages[7]. It was designed back in 2012 by a group of researchers Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah.

Language is well-known for core features like dynamic typing and multiple dispatch, also for a rich package management, support for scalable parallel and distributed computing, and many more. It is implemented mainly in Julia itself on top of the C, C++, Scheme and LLVM building blocks. In addition, it is garbage collected and employs Just In Time(JIT) compiler[1].

Introduction to a Julia Garbage Collector

Julia's Garbage Collector(GC) is *precise, generational, non-compacting, non-concurrent, non-parallel(single threaded), non preemptible* and utilizes *mark and sweep* algorithms. Main GC file for a reference is a GC.c under a src directory of a Julia codebase.

In general, Julia GC starts earlier than an actual memory overflow and in a multithreaded environment requires all the threads be in a safepoint, meaning to have a *precise* state. Here precise means that a garbage collector can track all the pointers to all the operated objects via a rootset which comprises CPU registers, Function Runtime Stack and Virtual Method Static Table[8,9]. Unless otherwise, allocating threads will halt before executing a garbage collection to other threads that have not yet reached a precise root set. For this purpose either all the threads should manipulate a memory at some point of their lifetime or should manually call a `GC.safepoint()` checkpoint. What it does is to invoke a segmentation fault by loading from a protected memory address. By the time it will be served by an OS, GC will be able to identify a rootset for that particular thread and to start a garbage collector.

Generational means that a garbage collector will divide an object pool into “young” and “old” generations following a hypothesis that approximately 80-90% recently allocated objects are unreachable or die soon. This pattern is also called “infant mortality”or “weak-generational hypothesis”[8, 10, 11-13]. Accordingly, GC first marks and sweeps inside a young pool, if no place is releasable, then mark and sweep goes through a whole allocated memory space. Nevertheless, as young objects might be also referred to from an older generation, “remembering set” should be implemented as well. Hence, Julia is expected to call a GC two times, first for automated half GC with a young generation check, then second time for a full GC if the need arises. Young objects that were referenced multiple GC cycles with more than `PROMOTE_AGE+1` are considered survived and tagged “old”.

With a non-compacting scheme Julia does not reorder allocated objects to resolve a memory or heap fragmentation issue. This might be a good starting point for many design discussions, nevertheless as compaction comes with additional latency and complexity on top of the classical mark and sweep, it is not implemented with Julia GC.

As Julia GC is *non-concurrent* unlike reference counting counterparts, for *mark and sweep* phase it stops all the threads, checks for safepoint, then starts marking, later sweeping and finalizing if necessary. The duration when all the threads are halted is called “stop the world” latency and will be discussed more in the Results and Discussions section along with simple examples and benchmarks results. The reason behind this is that it might be the source of many performance issues in correspondence with the application quality constraints. Nevertheless, as Julia GC implements the generational scheme, with younger generations “stop the world” latencies are expected to be short given the execution stage and dynamics of the program. On the other hand, as it does not check and clear the heap completely, many unused objects still might trash the memory and result in a memory overflow eventually.

Apart from a classical mark and sweep steps, Julia GC employs an interesting finalizer phase. With this stage some random portion of the unmarked objects might still stay at memory pull, however it will not be visible to the application but GC. Accordingly, they will not be erased by the current sweep stage, rather might be removed at the next sweep phase. This flexibility is given to support the scenarios when some objects might be requested at the later cycles. In

those cases, they can be restored from a finalizer by inserting another mutator code than application threads.

Problem Statement

Garbage collection and its performance in a multithreaded environment was and continues to be an important source of debates. Some authors argue that with increasing parallelism, garbage collectors get congested and application is bottlenecked[15, 17, 18, 19, 20]. Possible reasons are increased object lifespan and allocation capacity over multiple threads. According to early observations by Agesen, 1998, the program in average spends 20% of its execution time with garbage collection[20]. Recent findings of Gidra et.al in 2011 shows that OpenJDK7 GC might take up to 33% of the execution time with 48 cores as well[15]. On the other hand, latest research by both the same authors and beyond claim that observed bottleneck is not always relatable with the number of allocated objects and are providing some solutions to solve this challenge[14, 16]. The respective solutions are exploring balanced load distribution across NUMA memory architecture[14], while others are paying attention to simpler thread scheduling mechanisms[16]. Nevertheless, most of the GC research is done within a JVM environment, hence whatever is the possible reasons of a bottleneck, exploring a newly developed Julia GC under a scalable parallel abode will be an interesting challenge to attempt. Hence, this project aims to

- To inspect Julia GC in a multithreaded programs;
- To observe which certain applications are sensible to GC pauses;
- To understand what exact GC phases are responsible for long “stop the world” latencies.

Given Julia’s increasing popularity among distributed computing for a diverse set of computations starting from data science towards financial analysis, climate simulations and many more, inspection results might be useful to understand state of the art Julia GC better and to direct possible implementation changes.

Tracing Tools

BPFTrace and DTrace

DTrace (Solaris originally) and bpftrace (Linux) are tools that enable lightweight instrumentation or tracing of processes.

With BPFTrace it is possible to develop tracing scripts that will monitor any event. It could be both at the kernel, also at user or so called application space. Here, an event could be a system call, a function call, or something that happens within a single call. It could also be a timer or a hardware event[22]. With tracing tools one can write different response instructions to these events like yielding some statistics, or executing another shell process and many more[22]. It is feasible to track the system level resource utilization as well.

Bpftrace is a low overhead profiling tool and this is what makes it interesting. One doesn't need to restart applications, or recompile them or add print statements to debug in the source code of the target application. So this is very useful to debug systems where compilation is a challenge [22].

Uprobe vs USDT

The following table illustrates general classification of tracing tools[5] :

	kernel	userland
static	tracepoints	USDT* probes
dynamic	kprobes	uprobes

Here, USDT stands for Userland Statically Defined Tracing.

Generally, static tracing tools are compile time flags that could be inserted to the code and used to trigger a start of the data tracing or to end it. Here is how it is inserted into a GC.c compiler:

```
JL_PROBE_GC_BEGIN(collection);
```

Ways how to actually enable it with an entire codebase will be explained later below.

On the other hand, dynamic tracing tools listen for an entry or return of exported function calls via dynamic library. Here is how certain function is exported in Julia GC:

```
JL_DLLEXPORT void jl_gc_collect(jl_gc_collection_t collection)
```

Uprobe results for a test program:

Below is an allocating function that runs indefinitely until halted and creates an array of size N bytes:

```
function allocator(range)
    while true
        N = rand(range)
        buf = Array{UInt8}(undef, N)
    end
end[5]
```

The results could be replicated by running this command in one terminal. Here we do load julia source code and execute it with the given arguments:
`julia -L allocator.jl -e "allocator(64:128)"`

Then one needs to run these commands in a second terminal:

```
ubuntu@euca-10-1-4-194:~/julia$ sudo bpftrace -e
"uprobe:usr/lib/libjulia-internal.so:jl_gc_alloc { @[pid] =
hist(arg1); }"
Attaching 1 probe...
^C
```

```
@[19075]:
[64, 128)          100857 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@                        |
[128, 256)         182855 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
```

This histogram shows memory allocation distribution by process ID. This can trigger all the processes using the library, but it is also possible to trigger only by PID with `-p PID`. It is observable that though the function is called for a range 64:128 and major allocations are expected to happen between 64:128, they are in the range 128:256. This is because along with the array allocation there are some meta objects allocations happening in the heap that are increasing the total size of allocations.

USDT probes results for a test program:

Enabling support
We need to build julia by creating a Make.user file and adding **WITH_DTRACE=1** to enable USDT probes support[6].

Available probes are located in uprobes.d file under a directory `/src`:
julia:gc__begin: This is where GC starts checking for the safepoints.
julia:gc__stop_the_world: This marks the time when all the threads are at safe-point allowing GC to begin.
julia:gc__mark__begin: This is the beginning of a mark phase.
julia:gc__mark_end(scanned_bytes, perm_scanned): Here mark phase ends.
julia:gc__sweep_begin(full): This marks the beginning of a sweep phase.
julia:gc__sweep_end(): This marks the end of a sweep phase.
julia:gc__end: All other threads continue to work after GC is ended here.
julia:gc__finalizer: This is where the initial GC thread stops running the finalizers[6].

```
using Base.Threads

fib(x) = x <= 1 ? 1 : fib(x-1) + fib(x-2)

beaver = @spawn begin
    while true
```

```
        fib(30)
        GC.safepoint()
    end
end

allocator = @spawn begin
    while true
        zeros(1024)
    end
end

wait(allocator)
```
[6]

We run this using `julia -t 2` (This means we are using 2 threads). Here `@spawn` allocates a task to an available thread and executes it.

```
ubuntu@euca-10-1-4-194:~/julia$ sudo bpftrace --usdt-file-activation
contrib/bpftrace/gc_stop_the_world_latency.bt
Attaching 4 probes...
Tracing Julia GC Stop-The-World Latency... Hit Ctrl-C to end.
^C

@usecs[16514]:
[512, 1K)              1 |                                                    |
[1K, 2K)             105 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[2K, 4K)              65 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@                    |
```

This histogram shows latency distribution to reach stop stop-the-world phase in the executed Julia process.

In the current versions of Julia, GC.safepoint() should be called from a non-allocating thread to make sure that all threads are checked for a precise state otherwise the process would never yield to GC. Manually adding GC.safepoint() triggers segmentation fault by loading from a protected memory space under the hood and forces GC to execute. Future Julia versions are expected to include built-in function entry GC.safepoints().

Julia GC Benchmarking Results

General Information

The following h2o benchmark was used to inspect for a garbage collection performance - https://github.com/d-netto/db-benchmark/tree/master/juliadf. It is a collection of programs that employ well-known languages used in data science as Python, R, and Julia, also to execute popular database tools like Polars, Pandas, Spark and many more[23].

To begin with, first a 0.5GB of data is generated using the R script given in the readme file of the repository. The generated data includes nine sub-groups like `v1` to `v3`, and `id1` to `id6`. The

purpose is to run various benchmarks on this data and compare latencies of various GC phases when running these benchmarks individually.

To run the benchmarks, script present in **db-benchmark/juliadf/groupby-juliadf.jl** is run**.** With every benchmark execution, only a single sub benchmark is run. The reason behind is that because of a generational Julia GC, memory was quickly overloaded over time, but not cleared totally. Hence with the new sub-benchmark, memory overflow was happening. To solve this problem, a single sub-benchmark was run with every experimental execution.

At the same time when these benchmarks are running, one should run the bpftrace using various available Julia probes to see latencies of various Julia phases. The main goal is to check these latencies for multi-threaded programs so all these benchmarks are run with 2 threads. 4 and 8 threads were not tested as the underlying core was limited to two cores also.

Here is the script for a reference:
bench_tasks_vec = [
 # BenchTask("median v3 sd v3 by id4 id5", x -> combine(groupby(x, [:id4, :id5]), :v3 => median◦skipmissing => :median_v3, :v3 => std◦skipmissing => :sd_v3)),
 # BenchTask("max v1 - min v2 by id3", x -> combine(groupby(x, :id3), [:v1, :v2] => ((v1, v2) -> maximum(skipmissing(v1))-minimum(skipmissing(v2))) => :range_v1_v2)),
BenchTask("largest two v3 by id6", x -> combine(groupby(dropmissing(x, :v3), :id6), :v3 => (x -> partialsort!(x, 1:min(2, length(x)), rev=false)) => :largest2_v3)),
 # BenchTask("sum v3 count by id1:id6", x -> combine(groupby(x, [:id1, :id2, :id3, :id4, :id5, :id6]), :v3 => sum◦skipmissing => :v3, :v3 => length => :count))
]

Reproducing results

To reproduce the results, first in one terminal run this (every time comment out necessary sub-benchmark):

```
julia -t db-benchmark/juliadf/groupby-juliadf.jl
```

And in another terminal run one of the following:

Benchmark 1: BenchTask("median v3 sd v3 by id4 id5")

```
ubuntu@euca-10-1-4-194:~/julia$ sudo bpftrace --usdt-file-activation
contrib/bpftrace/gc_all.bt
Attaching 10 probes...
Tracing Julia GC Times... Hit Ctrl-C to end.
^C
```

// Latency distribution for a finalizer GC

```
@finalizer[27992]:
[4, 8)                 5 |@@@@@                                               |
```

```
[8, 16)                21 |@@@@@@@@@@@@@@@@@@@@@@                              |
[16, 32)               52 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[32, 64)                2 |@@                                                    |
```

// Latency distribution for a actual GC

```
@gc_phase_usecs[27992]:
[32K, 64K)             25 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@                       |
[64K, 128K)            52 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[128K, 256K)            3 |@@@                                                   |
```

// Latency distribution for a actual GC+safepoint check duration

```
@gc_total_usecs[27992]:
[32K, 64K)             25 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@                       |
[64K, 128K)            52 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[128K, 256K)            3 |@@@                                                   |
```

// Latency distribution for GC marking phase

```
@mark_usecs[27992]:
[16, 32)                2 |@@                                                    |
[32, 64)               18 |@@@@@@@@@@@@@@@@@@@                                   |
[64, 128)              47 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[128, 256)              3 |@@@                                                   |
[256, 512)              0 |                                                      |
[512, 1K)               0 |                                                      |
[1K, 2K)                0 |                                                      |
[2K, 4K)                0 |                                                      |
[4K, 8K)                4 |@@@@                                                  |
[8K, 16K)               6 |@@@@@@                                                |
[16K, 32K)              0 |                                                      |
[32K, 64K)              0 |                                                      |
[64K, 128K)             0 |                                                      |
[128K, 256K)            3 |@@@                                                   |
```

// Latency distribution for threads to reach “stop the world” or till GC is actually started to executed since it is called

```
@stop_the_world_usecs[27992]:
[8, 16)                 3 |@@@@                                                  |
[16, 32)                3 |@@@@                                                  |
[32, 64)                7 |@@@@@@@@@@@                                           |
[64, 128)              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[128, 256)             28 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@          |
[256, 512)              6 |@@@@@@@@@@                                            |
[512, 1K)               1 |@                                                     |
```

// Latency distribution for GC sweeping phase

```
@sweep_usecs[27992]:
[2K, 4K)                3 |@@@                                                   |
[4K, 8K)                0 |                                                      |
[8K, 16K)               0 |                                                      |
[16K, 32K)              0 |                                                      |
[32K, 64K)             28 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@                  |
[64K, 128K)            52 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
```

```
┌ Info: benchmark
│   mean(times) = 1.32781132542
└   mean(gctimes) = 0.045970915380000005
```

BenchMark 2: BenchTask("max v1 - min v2 by id3")

```
ubuntu@euca-10-1-4-194:~/julia$ sudo bpftrace --usdt-file-activation
```

```
contrib/bpftrace/gc_all.bt
Attaching 10 probes...
Tracing Julia GC Times... Hit Ctrl-C to end.
^C

@finalizer[28072]:
[4, 8)                 3 |@@@@                                                |
[8, 16)                6 |@@@@@@@@                                            |
[16, 32)              37 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[32, 64)               0 |                                                    |
[64, 128)              0 |                                                    |
[128, 256)             0 |                                                    |
[256, 512)             0 |                                                    |
[512, 1K)              0 |                                                    |
[1K, 2K)               0 |                                                    |
[2K, 4K)               0 |                                                    |
[4K, 8K)               0 |                                                    |
[8K, 16K)              1 |@                                                   |

@gc_phase_usecs[28072]:
[16K, 32K)            38 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[32K, 64K)             4 |@@@@@                                               |
[64K, 128K)            1 |@                                                   |
[128K, 256K)           3 |@@@@                                                |
[256K, 512K)           1 |@                                                   |

@gc_total_usecs[28072]:
[16K, 32K)            38 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[32K, 64K)             4 |@@@@@                                               |
[64K, 128K)            1 |@                                                   |
[128K, 256K)           3 |@@@@                                                |
[256K, 512K)           1 |@                                                   |

@mark_usecs[28072]:
[16, 32)               2 |@@@@                                                |
[32, 64)              22 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[64, 128)             12 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@                    |
[128, 256)             0 |                                                    |
[256, 512)             0 |                                                    |
[512, 1K)              0 |                                                    |
[1K, 2K)               0 |                                                    |
[2K, 4K)               1 |@@                                                  |
[4K, 8K)               4 |@@@@@@@@@@                                          |
[8K, 16K)              5 |@@@@@@@@@@@@@                                       |
[16K, 32K)             0 |                                                    |
[32K, 64K)             0 |                                                    |
[64K, 128K)            0 |                                                    |
[128K, 256K)           5 |@@@@@@@@@@@@@                                       |



@stop_the_world_usecs[28072]:
[8, 16)                3 |@@@@@                                               |
[16, 32)               1 |@                                                   |
[32, 64)               4 |@@@@@@                                              |
[64, 128)             31 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[128, 256)             8 |@@@@@@@@@@@@@@                                      |

@sweep_usecs[28072]:
[2K, 4K)               3 |@@@                                                 |
[4K, 8K)               0 |                                                    |
[8K, 16K)              1 |@                                                   |
```

```
[16K, 32K)             41 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[32K, 64K)              5 |@@@@@@                                                      |
[64K, 128K)             1 |@                                                           |
```

```
┌ Info: benchmark
│   mean(times) = 1.30983484848
└   mean(gctimes) = 0.00858797214
```

BenchMark 3: BenchTask("largest two v3 by id6")

```
ubuntu@euca-10-1-4-194:~/julia$ sudo bpftrace --usdt-file-activation
contrib/bpftrace/gc_all.bt
Attaching 10 probes...
Tracing Julia GC Times... Hit Ctrl-C to end.
^C

@finalizer[28102]:
[4, 8)                 15 |@@@@@@@@@@@@@                                             |
[8, 16)                 5 |@@@@                                                      |
[16, 32)               63 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[32, 64)                0 |                                                          |
[64, 128)               4 |@@@                                                       |
[128, 256)              0 |                                                          |
[256, 512)              0 |                                                          |
[512, 1K)               0 |                                                          |
[1K, 2K)                0 |                                                          |
[2K, 4K)                0 |                                                          |
[4K, 8K)                0 |                                                          |
[8K, 16K)               1 |                                                          |
[16K, 32K)              1 |                                                          |

@gc_phase_usecs[28102]:
[16K, 32K)              1 |@                                                         |
[32K, 64K)             45 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[64K, 128K)            39 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@        |
[128K, 256K)            2 |@@                                                        |
[256K, 512K)            2 |@@                                                        |

@gc_total_usecs[28102]:
[16K, 32K)              1 |@                                                         |
[32K, 64K)             37 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     |
[64K, 128K)            40 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[128K, 256K)            8 |@@@@@@@@@@@                                               |
[256K, 512K)            3 |@@@                                                       |

@mark_usecs[28102]:
[16, 32)                1 |                                                          |
[32, 64)                7 |@@@@@                                                     |
[64, 128)              67 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[128, 256)              0 |                                                          |
[256, 512)              0 |                                                          |
[512, 1K)               0 |                                                          |
[1K, 2K)                0 |                                                          |
[2K, 4K)                2 |@                                                         |
[4K, 8K)                6 |@@@@                                                      |
[8K, 16K)               4 |@@@                                                       |
[16K, 32K)              0 |                                                          |
[32K, 64K)              0 |                                                          |
[64K, 128K)             0 |                                                          |
[128K, 256K)            2 |@                                                         |
```

```
@stop_the_world_usecs[28102]:
[32, 64)              22 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[64, 128)             10 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[128, 256)            19 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[256, 512)             1 |@@                                                              |
[512, 1K)              0 |                                                                |
[1K, 2K)               1 |@@                                                              |
[2K, 4K)               3 |@@@@@@@@                                                        |
[4K, 8K)               1 |@@                                                              |
[8K, 16K)             14 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[16K, 32K)             4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[32K, 64K)             3 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[64K, 128K)           11 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|

@sweep_usecs[28102]:
[8K, 16K)              1 |@                                                               |
[16K, 32K)             1 |@                                                               |
[32K, 64K)            46 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[64K, 128K)           40 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[128K, 256K)           0 |                                                               |
[256K, 512K)           1 |@                                                               |

┌ Info: benchmark
│   mean(times) = 1.91489067545
└   mean(gctimes) = 0.0474870233
```

BenchMark 4: BenchTask("sum v3 count by id1:id6")

```
ubuntu@euca-10-1-4-194:~/julia$ sudo bpftrace --usdt-file-activation
contrib/bpftrace/gc_all.bt
Attaching 10 probes...
Tracing Julia GC Times... Hit Ctrl-C to end.
^C

@finalizer[27557]:
[16, 32)              88 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[32, 64)              16 |@@@@@@@@@@                                                      |
[64, 128)              3 |@                                                               |
[128, 256)             0 |                                                                |
[256, 512)             0 |                                                                |
[512, 1K)              0 |                                                                |
[1K, 2K)               0 |                                                                |
[2K, 4K)               0 |                                                                |
[4K, 8K)               0 |                                                                |
[8K, 16K)              1 |                                                                |
[16K, 32K)             1 |                                                                |

@gc_phase_usecs[27557]:
[32K, 64K)            77 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[64K, 128K)           28 |@@@@@@@@@@@@@@@@@@                                              |
[128K, 256K)           3 |@@                                                              |
[256K, 512K)           1 |                                                                |

@gc_total_usecs[27557]:
[32K, 64K)            64 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[64K, 128K)           41 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@                            |
[128K, 256K)           3 |@@                                                              |
[256K, 512K)           1 |                                                                |

@mark_usecs[27557]:
[32, 64)              14 |@@@@@@@@@                                                       |
```

```
[64, 128)              81 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[128, 256)              1 |                                                          |
[256, 512)              0 |                                                          |
[512, 1K)               0 |                                                          |
[1K, 2K)                0 |                                                          |
[2K, 4K)                3 |@                                                         |
[4K, 8K)                4 |@@                                                        |
[8K, 16K)               4 |@@                                                        |
[16K, 32K)              0 |                                                          |
[32K, 64K)              0 |                                                          |
[64K, 128K)             0 |                                                          |
[128K, 256K)            2 |@                                                         |

@stop_the_world_usecs[27557]:
[32, 64)                7 |@@@@@@@                                                   |
[64, 128)              50 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[128, 256)             28 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@                          |
[256, 512)              1 |@                                                         |
[512, 1K)               0 |                                                          |
[1K, 2K)                0 |                                                          |
[2K, 4K)                0 |                                                          |
[4K, 8K)                2 |@@                                                        |
[8K, 16K)              10 |@@@@@@@@@@                                                |
[16K, 32K)             11 |@@@@@@@@@@@                                               |

@sweep_usecs[27557]:
[32K, 64K)             78 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[64K, 128K)            31 |@@@@@@@@@@@@@@@@@@@@@@@@                                  |

┌ Info: benchmark
│   mean(times) = 2.00562935772
└   mean(gctimes) = 0.05107096632
```

Julia GC Benchmarking Discussions

Following discussions are the main points obtained from the results:

1) First and the second pair of sub-benchmarks have similar performance. This could be because each of the pair combining the database groups similarly. First and second groups employ the simpler algorithms, the third and fourth ones use more complex data manipulation via sorting and counting on the data allocation critical path.

2) Sweeping phase generally takes more time than the marking phase. The possible explanation might be that freeing the memory will require more time than just marking. As with marking only a single bit is set, but with sweeping different object allocations should be released.

3) GC spends much time for safepoint checking itself other than main GC duties. Although this does not always happen, there are still a couple of times in a distribution when GC safepoint checks go up to 32k usecs from average 64 usec, for example from a last sub-benchmark. Other than that there are many examples when gc latency is around 0.03 to 0.06 seconds on average or goes as high as 0.256 seconds. However, if we compare this with a video with 30 frames per second, which executes one frame in

around 0.033 second, this stops the world latency and/or general GC latency is around 2 video games frame long. It is very performance taxing and will even might affect the user experience and quality of service**.** Therefore**,** this high stop-the-world latency is a big issue persisting in julia which needs to be worked on. This example also makes it clear that it will be challenging to develop video games in julia as they would end up being too laggy.

Conclusion

To conclude, GC latency is critical for real time applications including gaming, robotics, avionics and related, hence should be improved within Julia GC framework too. Also, to solve the issue one should look at optimizing a “stop the world” stage and sweeping phase. In addition, multithreaded GC would be useful to implement, as the current state of the art Julia GC is non-parallel and single threaded. However, again it is important to be careful with distributing GC tasks in parallel to avoid other possible bottlenecks or data races as was being discussed in many other related parallel GC research. Here parallel GC is referred not as running GC in parallel or concurrently with mutator threads, instead it is for executing GC alone in multiple threads.

References

[1] Wikipedia, n.d., “Julia Programming Language”
https://en.wikipedia.org/wiki/Julia_(programming_language)#History
[2] A. Amerio, 2019, “Code optimization in Julia”. Last modified October 26, 2019.
https://techytok.com/code-optimisation-in-julia/
[3] BPFtrace GitHub Repo, “bpftrace Reference Guide”. Last modified September 1, 2021.
https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md#2---basic-variables
[4] Julia GitHub Repo, “The Julia Language”. Last modified December 7, 2021. https://github.com/JuliaLang/julia
[5] V. Churavy, 2021. “BPFTrace and Julia”. Last modified October 3, 2021.
https://vchuravy.dev/notes/2021/08/bpftrace/
[6] Julia Documentation, n.d..“Instrumenting Julia with DTrace, and bpftrace”.
https://docs.julialang.org/en/v1.8-dev/devdocs/probes/#Available-probes
[7] J. Bezanson, S. Karpinski, V.B. Shah, A. Edelman, 2012. “Why We Created Julia".
https://julialang.org/blog/2012/02/why-we-created-julia/
[8] Daniil Berezun and Dmitri Boulytchev. 2014. “Precise garbage collection for C++ with a non-cooperative compiler”. *In Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR '14).* Association for Computing Machinery, New York, NY, USA, Article 15, 1–8.
DOI:https://doi.org/10.1145/2687233.2687244
[9] C. Krintz, 2021. *UCSB*, *Fall 21, CS-263 Lecture Slides*.
https://sites.cs.ucsb.edu/~ckrintz/classes/f21/cs263/sched.html
[10] Wikipedia, n.d.. “Garbage collection (computer science)”. Last accessed December 7, 2021.
https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
[11] A. W. Appel. *Simple generational garbage collection and fast allocation*. SP&E, 19(2):171–183, 1989.
[12] H. Lieberman, C. Hewitt. *A real-time garbage collector based on the lifetimes of objects.* CACM, 26(6):419–429, 1983.
[13] D. Ungar. *Generation scavenging: A non-disruptive high performance storage reclamation algorithm.* In SDE ’84, pages 157–167. ACM, 1984.
[14] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, et al. *“*A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores”. *In ASPLOS 13-Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 229–240, 2013.
https://hal.inria.fr/hal-00868012/document

[15] L. Gidra, G. Thomas, J. Sopena, M. Shapiro. “Assessing the scalability of garbage collectors on many cores”. *In SOSP Workshop on Programming Languages and Operating Systems, PLOS ’11*, pages 1–5. ACM, 2011
[16] Junjie Qian, Witawas Srisa-an, Sharad Seth, Hong Jiang, Du Li, and Pan Yi. 2016. “Exploiting FIFO Scheduler to Improve Parallel Garbage Collection Performance”. *In Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '16)*. Association for Computing Machinery, New York, NY, USA, 109–121. DOI:https://doi.org/10.1145/2892242.2892248
[17] K. Du Bois, J. B. Sartor, S. Eyerman, L. Eeckhout. *Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications.* In ACM SIGPLAN Notices, volume 48, pages 355–372. ACM, 2013.
[18] J. Qian, D. Li, W. Srisaan, H. Jiang, S. Seth. “Factors Affecting Scalability of Multithreaded Java Applications on 120 Manycore Systems”. *In 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 167–168. IEEE, 2015.
[19] J. Qian, W. Srisa-an, D. Li, H. Jiang, S. Seth. “SmartStealing: Analysis and Optimization of Work Stealing in Parallel Garbage Collection for Java VM”. *In Proceedings of the 12th International Conference on Principles and Practice of Programming in Java*, pages 170–181. ACM, 2015.
[20] O. Agesen, “GC Points in a Threaded Environment”, Sun Microsystems, https://dl.acm.org/doi/pdf/10.5555/974974
[21] https://github.com/d-netto/db-benchmark
[22] H. Lai, 2019, “Full-system dynamic tracing on Linux using eBPF and bpftrace” Last modified Jan 31, 2019. https://www.joyfulbikeshedding.com/blog/2019-01-31-full-system-dynamic-tracing-on-linux-using-ebpf-and-bpftrace.html#dtrace-the-father-of-tracing
[23] M. Dowle, 2021. “Database-like ops benchmark”. https://h2oai.github.io/db-benchmark/