Inspection of Julia GC for multithreaded programs

Nazerke Turtayeva
Deept Mahendiratta

CS 263, Runtime Systems
Project Presentation

UC SANTA BARBARA

Contents

Intro to Julia

What: High-level, high-performance, dynamic programming language - aimed to employ the best of both worlds

Who: GP, but mainly for scientific community - numerical analysis
~10,000 companies, 1500 universities[1]

When: 2012, MIT

Features: Multiple dispatch, dynamic typing, parallel and distributed computing, no need for a vectorized code, package management

Compiler: Written in Julia, C/C++, JIT

vs

vs

Syntax in short

Q: What makes Julia to perform close to C?

A: (1) Julia features + (2) Efficient user code

Features:

- Dynamic type inferring

```
d = Int[1,2,3,4,5]   vs   d = [1,2,3,4,5]
```

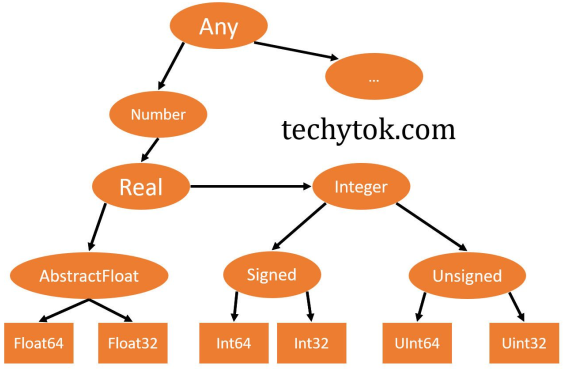
- Multiple dispatch (don't overuse)
- Using function profiling

User:

- Type stability to infer types beforehands correctly

  NO `d = 1 ... d = 2.3 ...  d = [1,2,3,4,5]`

- Common sense: no global vars, static arrays, @inbounds, etc[2]
- Identifying bottlenecks



Types hierarchy in Julia [2]

```
function test1(x::Int)
    println("$x is an Int")
end

function test1(x::Float64)
    println("$x is a Float64")
end

function test1(x)
    println("$x is neither an Int
nor a Float64")
end
```

Multiple dispatch example [2]

Motivation

Source of issues:

1) > Non parallel, mark and sweep GC
   > multi threading
   > more data to collect during “stop the world”(stw)
   > higher “stw” latency
   <u>> lower average performance</u>
2) > Some non-allocational threads might not never yield to a safe(precise) point
   > Hence, allocating threads might stall
   <u>> lower average performance</u>

Motivation

Solutions:

1) Inspecting the GC to identify which specific applications are more vulnerable to the "stw" latencies
2) Identifying which parts of the GC collections and sweeping relatable for this

GC in Julia

- Julia's garbage collector algorithm is called *mark and sweep.*
- It is precise, generational, non-compacting, non-parallel.
- The mark phase: Where all objects that are not garbage are found and marked so
- The sweep phase: where all unmarked objects are cleaned
- Starts earlier than memory overflow
- The top Julia GC source file is GC.c

Triggering manual GC

- When the garbage collector detects that the program can no longer access the object, then it will run the finalizer, and then collect (free) the object.

- Note that the garbage collector can still access the object, even though the program cannot.

Example of Manual GC

```
julia> r = Ref(0)
Base.RefValue{Int64}(0)

julia> finalizer(r) do r
       println(r)
       end
Base.RefValue{Int64}(0)

julia> r = nothing

julia> GC.gc()
Base.RefValue{Int64}(0)
```

DTrace and BPFTrace

- `DTrace`(Solaris originally) and `bpftrace`(Linux) are tools that enable **lightweight instrumentation** of processes.

- You can turn the instrumentation on and off while the process is running, and with instrumentation off the overhead is minimal.

Tracing interfaces

	kernel	userland
static	tracepoints	USDT* probes
dynamic	kprobes	uprobes

[5]

- We will focus on userland for now, since that is the most useful feature for understanding applications, like Julia
- USDT - **U**ser-level **S**tatically **D**efined **T**racing

DTrace and BPFTrace

- Using nm we can list all of the exported runtime functions of Julia.

```
nm -D /usr/lib/julia/libjulia-internal.so | grep jl_gc_
```

```
0000000000015215 T jl_gc_add_finalizer
0000000000015221 T jl_gc_add_finalizer_th
000000000001522d T jl_gc_add_ptr_finalizer
0000000000015239 T jl_gc_alloc
0000000000015245 T jl_gc_alloc_0w
0000000000015251 T jl_gc_alloc_1w
000000000001525d T jl_gc_alloc_2w
0000000000015269 T jl_gc_alloc_3w
0000000000015275 T jl_gc_allocobj
0000000000015281 T jl_gc_alloc_typed
000000000001528d T jl_gc_big_alloc
```

DTrace and BPFTrace

- An alternative way to find all valid `uprobe` is to use bpftrace:

```
sudo bpftrace -l 'uprobe:/usr/lib/julia/libjulia-internal.so:jl_gc_*'
```

```
uprobe:/usr/lib/julia/libjulia-internal.so:jl_gc_add_finalizer
uprobe:/usr/lib/julia/libjulia-internal.so:jl_gc_add_finalizer_th
uprobe:/usr/lib/julia/libjulia-internal.so:jl_gc_add_ptr_finalizer
uprobe:/usr/lib/julia/libjulia-internal.so:jl_gc_alloc
uprobe:/usr/lib/julia/libjulia-internal.so:jl_gc_alloc_0w
uprobe:/usr/lib/julia/libjulia-internal.so:jl_gc_alloc_1w
uprobe:/usr/lib/julia/libjulia-internal.so:jl_gc_alloc_2w
uprobe:/usr/lib/julia/libjulia-internal.so:jl_gc_alloc_3w
uprobe:/usr/lib/julia/libjulia-internal.so:jl_gc_alloc_page
uprobe:/usr/lib/julia/libjulia-internal.so:jl_gc_alloc_typed
uprobe:/usr/lib/julia/libjulia-internal.so:jl_gc_allocobj
uprobe:/usr/lib/julia/libjulia-internal.so:jl_gc_big_alloc
```

Using uprobe[5]

- The function allocator is going to run forever and allocate an array of size N bytes.

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

- Running it: **julia -L allocator.jl -e "allocator(64:128)"**

Results: uprobe

```
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 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
```

- Memory allocation distribution by process ID
- This triggers all the processes using the library, but it is possible to trigger only by PID with `-p PID`

Using USDT probes[6]

- Enabling support:
  Using `WITH_DTRACE=1` in Make.user file

- Available probes, `uprobes.d`:
  - **julia:gc__begin**: GC begins running on one thread and triggers stop-the-world.
  - **julia:gc__stop_the_world**: All threads have reached a safepoint and GC runs.
  - **julia:gc__mark__begin**: Beginning the mark phase
  - **julia:gc__mark_end(**scanned_bytes, perm_scanned): Mark phase ended
  - **julia:gc__sweep_begin**(full): Starting sweep
  - **julia:gc__sweep_end**(): Sweep phase finished
  - **julia:gc__end**: GC is finished, other threads continue work
  - **julia:gc__finalizer**: Initial GC thread has finished running finalizers

Using USDT probes

```
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)
```

- Run this using `julia -t 2` (This means we are using 2 threads)
- `@spawn` allocates a Task to an available thread and executes

Results: USDT probe for “stop the world”

```
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 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@                    |
```

- Latency distribution of the stop-the-world phase in the executed Julia process.

Contents of `gc_stop_the_world_latency.bt`

```
#!/usr/bin/env bpftrace

BEGIN
{
    printf("Tracing Julia GC Stop-The-World Latency... Hit Ctrl-C to end.\n");
}

usdt:usr/lib/libjulia-internal.so:julia:gc__begin
{
    @start[pid] = nsecs;
}

usdt:usr/lib/libjulia-internal.so:julia:gc__stop_the_world
/@start[pid]/
{
    @usecs[pid] = hist((nsecs - @start[pid]) / 1000);
    delete(@start[pid]);
}

END
{
    clear(@start);
}
```

Using USDT probes

```
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)
```

- `GC.safepoint()` should be called from a non-allocating thread to make sure that all threads are checked for a precise state.
- It triggers segmentation fault by loading from a protected memory space under the hood and forces GC to execute.
- Future Julia versions expected to include built-in function entry GC.safepoints()

Project challenges

Never use M1 chip Mac for GC tracing! :)

Using linux servers with sudo access is a prerequisite

References

[1] https://en.wikipedia.org/wiki/Julia_(programming_language)#History
[2] https://techytok.com/code-optimisation-in-julia/
[3] https://juliacomputing.com/
[4] https://github.com/JuliaLang/julia
[5] https://vchuravy.dev/notes/2021/08/bpftrace/
[6] https://docs.julialang.org/en/v1.8-dev/devdocs/probes/#Available-probes

UC SANTA BARBARA