Garbage Collector Interface for Julia

Authors: Diogo Correia Netto
Collaborators: Kiran Pamnany
Last Updated: 07/01/2024
Status: DRAFT

Introduction

Background

As part of RelationalAI's, JuliaHub's, and JuliaLab's investments in improving Garbage Collection (GC) performance in Julia, we have been collaborating with the MMTk (Memory Management Toolkit) team, led by Steve Blackburn and Tony Hosking from the Australian National University (ANU).

For development and testing purposes, the MMTk team has been maintaining a separate fork of Julia where changes in the compiler and runtime required to enable MMTk are guarded by conditional preprocessor clauses (i.e., C/C++ #ifdef). This is problematic because updating to every new Julia version requires a significant amount of refactoring in MMTk's fork. We believe that exposing a GC interface in Julia and simply letting MMTk implement this interface would alleviate some of this maintenance burden.

This design document outlines the changes to Julia's source tree and build system that we plan to implement to make MMTk a build-time option for Julia. This will enable us to:

- Accelerate the development velocity of the ANU team by relieving them of the burden of updating their code for each Julia release.
- Get feedback and bug reports from early adopters in the community about the current implementation, which will help us identify correctness and performance issues.
- Make the work done by the MMTk team tangible and concrete for the Julia community by providing a GC implementation that can be easily tested and benchmarked.

The ANU project is nearing a milestone where we want to start soliciting community input.

This work will prepare us to make MMTk easily available for early adopters in the community, enabling us to request feedback from them more efficiently.

Out of Scope

The goal of this project is solely to enable MMTk as a build-time option for Julia (i.e. no run-time/dynamic selection of a GC implementation). We are also not aiming to make MMTk the default GC implementation for Julia at this time.

In particular, note that Julia supports a variety of operating systems (e.g., MacOS, FreeBSD) and architectures (e.g., aarch64, PowerPC) that are not currently supported by MMTk, which

is only `Linux x86_64` at the moment. Supporting these other platforms is left as future work.

We expect that making MMTk the default GC implementation will involve porting it to these platforms, as well as addressing stability and performance issues discovered through testing by early adopters in the community.

Interface Design

The first portion of this work will consist of creating a GC interface in the Julia runtime so that any collector implementation conforming to the interface can plug into Julia. We expect the interface to be divided into the following modules:

- GC system-wide initialization.
- GC per-thread initialization.
- GC controls.
- GC metrics.
- Conservative & Foreign GC support.
- Allocation.
- Marking.
- Runtime write-barriers.
- Allocation Fast-Path & Codegen write-barriers.

Note that object finalization is not included in any of the modules above. Initially, we expect that any third-party GC implementing this interface will reuse the finalizer implementation from Julia (as MMTk does now).

The same holds for the stop-the-world and safe-point machinery: we expect that initially, any third-party GC will use the "`mprotect` a safepoint page & trap on `SEGV`" approach that the stock GC uses to stop threads for a collection.

Finally, note that we expect that initially, all barriers needed by a third-party GC will be write-barriers. This is currently the case for MMTk, and we expect it to remain so in the near future.

Prelude: GC TLS for Stock Julia and for Third-Party GCs

Nearly all GC functions access a limited number of fields in Julia's thread-local-storage. For the GC interface, we plan to encapsulate the fields frequently accessed by Julia's stock GC into a separate structure `jl_tls_states`.

```
typedef struct {
    jl_thread_heap_t heap;
    jl_gc_page_stack_t page_metadata_allocd;
    jl_thread_gc_num_t gc_num;
    jl_gc_markqueue_t mark_queue;
    jl_gc_mark_cache_t gc_cache;
    _Atomic(size_t) gc_sweeps_requested;
    arraylist_t sweep_objs;
} jl_gc_tls_states_t;
```

Figure 1: Stock Julia's GC TLS.

If a third-party GC is enabled, we will replace such a structure with another structure `jl_gc_tls_states_t` defined by the third-party GC in `<include/gc-tls.h`. This structure must be plain-old-data in order to be inlined in `jl_tls_states_t`.

```
#ifndef THIRD_PARTY_GC
// contains the definition of the GC TLS for the stock GC
#include "gc-stock-tls.h”
#else
// contains the definition of the GC TLS for the third-party GC
#include <include/gc-tls.h>
#endif

typedef struct _jl_tls_states_t {
  // Other fields…
   jl_gc_tls_states_t gc_tls;
   // Other Fields…
} jl_tls_states_t;
```

Figure 2: GC thread-local-state.

This pattern will enable us to retain the benefits of inlining the `heap`, `gc_num`, and other fields into Julia's TLS (note that codegen assumes certain fields of the heap structure are at fixed offsets from the TLS start).

A third-party GC could define `jl_gc_tls_states_t` in any way – a plain-old-data C structure, as the stock GC does, or a `void*`, to allow interoperation with GC implementations written in other languages.

System-wide Initialization

Functions that are called exactly once for every Julia process during initialization.

API

- `void jl_gc_init(void)`
  - Description: System-wide initialization function. Responsible for initializing global locks as well as global memory parameters (e.g. target heap size) used by the collector.
  - Required Annotations: None.
- `void jl_start_gc_threads(void)`
  - Spawns GC threads.
  - Required Annotations: None.

Per-Thread Initialization

Functions that are called exactly once for every Julia thread during their initialization.

API

- `void jl_gc_init_thread_heap(jl_ptls_t)`
  - Description: Initializes thread-local data structures such as thread-local object pools, thread-local remembered sets and thread-local allocation counters. Should be called exactly once per Julia thread.
  - Required Annotations: JL_NOTSAFEPOINT.
- `void jl_gc_destroy_thread_heap(jl_ptls_t)`
  - Description: Deallocates any memory previously used for thread-local GC data structures. Mostly used to ensure that we perform this memory cleanup for foreign threads that are about to leave Julia.
  - Required Annotations: None.

Controls

Functions to disable or enable the collector, to trigger a collection cycle, etc.

API

- `int jl_gc_enable(int)`
  - Description: Enables or disables (depending on the value of the argument) the collector. Returns whether GC was previously enabled.
  - Required Annotations: `JL_DLLEXPORT`.
- `int jl_gc_is_enabled(void)`
  - Description: Returns whether the collector is enabled.
  - Required Annotations: `JL_DLLEXPORT`.
- `void jl_gc_set_max_memory(uint64_t)`
  - Description: Sets a soft limit to Julia's heap.
  - Required Annotations: `JL_DLLEXPORT`.
- `int jl_gc_collect(jl_gc_collection_t)`
  - Description: Runs a GC cycle. This function's parameter determines whether we're running an incremental, full, or automatic (i.e. heuristic driven) collection. Returns whether we should run a collection cycle again (e.g. a full mark right after a full sweep to ensure we do a full heap traversal).
  - Required Annotations: `JL_DLLEXPORT`.
- `int jl_gc_is_collector_thread(int)`
  - Description: Returns whether the thread with a particular `tid` is a collector thread
  - Required Annotations: `JL_DLLEXPORT`
- `const char* jl_gc_active_impl(void)`
  - Description: Returns which GC implementation is being used and possibly its version according to the list of supported GCs. NB: it should clearly identify the GC by including e.g. 'stock' or 'mmtk' as a substring.
  - Required Annotations: `JL_DLLEXPORT`
- `void jl_gc_sweep_stack_pools_and_mtarraylist_buffers(jl_ptls_t)`
  - Description: Sweep Julia's stack pools and mtarray buffers. Note that this function has been added to the interface as each GC should implement it but it will most likely not be used by other code in the runtime. It still needs to be annotated with `JL_DLLEXPORT` since it is called from Rust by MMTk.
  - Required Annotations: `JL_DLLEXPORT`

Metrics

Functions that expose GC statistics to a Julia user.

This interface is messy as it has grown organically over the years. We hope to refactor all these (see https://github.com/JuliaLang/julia/pull/50772) in the near future.

API

- `jl_gc_num_t jl_gc_num(void)`
  - Description: Retrieves Julia's GC_Num (structure that stores GC statistics).
  - Required Annotations: `JL_DLLEXPORT`.
- `int64_t jl_gc_diff_total_bytes(void)`

    - Description: Returns the difference between the current value of total live bytes now (live bytes at the last collection plus number of bytes allocated since then), compared to the value at the last time this function was called.
    - Required Annotations: `JL_DLLEXPORT`.
- `int64_t jl_gc_sync_total_bytes(int64_t)`
    - Description: Returns the difference between the current value of total live bytes now (live bytes at the last collection plus number of bytes allocated since then) compared to the value at the last time this function was called. The offset parameter is subtracted from this value in order to obtain the return value.
    - Required Annotations: `JL_DLLEXPORT`.
- `int64_t jl_gc_pool_live_bytes(void)`
    - Description: Returns the number of pool allocated bytes. This could always return 0 for GC implementations that do not use pools.
    - Required Annotations: `JL_DLLEXPORT`.
- `int64_t jl_gc_live_bytes(void)`
    - Description: Returns the number of live bytes at the end of the last collection cycle (doesn't include the number of allocated bytes since then).
    - Required Annotations: `JL_DLLEXPORT`.
- `void jl_gc_get_total_bytes(int64_t*)`
    - Description: Stores the number of live bytes at the end of the last collection cycle plus the number of bytes we allocated since then into the 64-bit integer pointer passed as an argument.
    - Required Annotations: `JL_DLLEXPORT`.
- `uint64_t jl_gc_get_max_memory(void)`
    - Description: Retrieves the value of Julia's soft heap limit.
    - Required Annotations: `JL_DLLEXPORT`.
- `int jl_gc_total_hrtime(void)`
    - Description: High-resolution (nano-seconds) value of total time spent in GC.
    - Required Annotations: `None`.

Allocation

Functions to perform dynamic memory allocation of small, large and permanently allocated objects.

API

- `jl_value_t *jl_gc_small_alloc(jl_ptls_t, int, int, jl_value_t*)`
    - Description: Allocates small objects and increments Julia allocation counters. The (possibly unused in some implementations) offset to the arena in which we're allocating is passed in the second parameter, and the object size in the third parameter. If thread-local allocators are used, then this function should allocate in the thread-local allocator of the thread referenced by the `jl_ptls_t` argument. An additional (last) parameter containing information about the type of the object being allocated may be used to record an allocation of that type in the allocation profiler.
    - Required Annotations: `JL_DLLEXPORT`.
- `jl_value_t *jl_gc_big_alloc(jl_ptls_t, size_t, jl_value_t*)`
    - Description: Allocates large objects and increments Julia allocation counters. If thread-local allocators are used, then this function should allocate in the thread-local allocator of the thread referenced by the `jl_ptls_t` argument. An additional (last) parameter containing information about the type of the

object being allocated may be used to record an allocation of that type in the allocation profiler.
    - Required Annotations: `JL_DLLEXPORT`.
- `void *jl_gc_counted_malloc(size_t)`
    - Description: Wrapper around Libc `malloc` that updates Julia allocation counters.
    - Required Annotations: `JL_DLLEXPORT`.
- `void *jl_gc_counted_calloc(size_t, size_t)`
    - Description: Wrapper around Libc `calloc` that updates Julia allocation counters.
    - Required Annotations: `JL_DLLEXPORT`.
- `void jl_gc_counted_free_with_size(void*, size_t)`
    - Description: Wrapper around Libc `free` that updates Julia allocation counters.
    - Required Annotations: `JL_DLLEXPORT`.
- `void *jl_gc_counted_realloc_with_old_size(void*, size_t, size_t)`
    - Description: Wrapper around Libc `realloc` that updates Julia allocation counters.
    - Required Annotations: `JL_DLLEXPORT`.
- `void *jl_gc_managed_malloc(size_t)`
    - Description: Wrapper around Libc `malloc` that's used to dynamically allocate memory for Arrays and Strings. It increments Julia allocation counters and should check whether we're close to the Julia heap target, and therefore, whether we should run a collection.

      Note that this doesn't record the size of the allocation request in a side metadata (i.e. a few words in front of the memory payload): this function is used for Julia object allocations, and we assume that there is already a field in the Julia object being allocated that we may use to store the size of the memory buffer.

    - Required Annotations: `JL_DLLEXPORT`.
- `jl_weakref_t *jl_gc_new_weakref_th(jl_ptls_t, jl_value_t*)`
    - Description: Allocates a new weak-reference, assigns its value and increments Julia allocation counters. If thread-local allocators are used, then this function should allocate in the thread-local allocator of the thread referenced by the first `jl_ptls_t` argument.
    - Required Annotations: `JL_DLLEXPORT`.
- `jl_weakref_t *jl_gc_new_weakref(jl_value_t*)`
    - Description: Allocates a new weak-reference, assigns its value and increments Julia allocation counters. If thread-local allocators are used, then this function should allocate in the thread-local allocator of the current thread.
    - Required Annotations: `JL_DLLEXPORT`.
- `jl_value_t *jl_gc_allocobj(size_t)`
    - Description: Allocates an object whose size is specified by the function argument and increments Julia allocation counters. If thread-local allocators are used, then this function should allocate in the thread-local allocator of the current thread.
    - Required Annotations: `JL_DLLEXPORT`.
- `void *jl_gc_perm_alloc(size_t, int, unsigned, unsigned)`
    - Description: Permanently allocates a memory slot of the size specified by the first parameter. This block of memory is allocated in an immortal region that is never swept. The second parameter specifies whether the memory should be filled with zeros. The third and fourth parameters specify the alignment

and an offset in bytes, respectively. Specifically, the pointer obtained by advancing the result of this function by the number of bytes specified in the fourth parameter will be aligned according to the value given by the third parameter in bytes.
    - Required Annotations: `JL_NOTSAFEPOINT`.
- `jl_value_t *jl_gc_permobj(size_t, int, unsigned, unsigned)`
    - Description: Permanently allocates an object of the size specified by the first parameter. Size of the object header must be included in the object size. This object is allocated in an immortal region that is never swept. The second parameter specifies the type of the object being allocated and will be used to set the object header.
    - Required Annotations: `JL_NOTSAFEPOINT`.
- `void jl_gc_notify_image_load(const char*, size_t)`
    - Description: Notifies the GC about memory addresses that are set when loading the boot image. The GC may use this information to, for instance, determine that such objects should be treated as marked and belong to the old generation in nursery collections.

Runtime Write-Barriers

Functions responsible for triggering a GC write-barrier in field assignments – where the assigned value is a pointer – performed in the Julia runtime.

API

- `void jl_gc_queue_root(const jl_value_t*)`
    - Description: Write barrier slow-path. If a generational collector is used, it may enqueue an old object into the remembered set of the calling thread.
    - Required Annotations: `JL_DLLEXPORT`.
- `void jl_gc_queue_multiroot(const jl_value_t*, const void*, jl_datatype_t *)`
    - Description: In a generational collector, this function is used to walk over the fields of the object specified by the second parameter (as defined by the data type in the third parameter). If a field points to a young object, the first parameter is enqueued into the remembered set of the calling thread.
    - Required Annotations: `JL_DLLEXPORT`.
- `void jl_gc_wb_back(const void*)`
    - Description: If a generational collector is used, checks whether the function argument points to an old object, and if so, calls the write barrier slow path above. In most cases, this function is used when its caller has verified that there is a young reference in the object that's being passed as an argument to this function.
    - Required Annotations: `STATIC_INLINE`, `JL_NOTSAFEPOINT`.
- `void jl_gc_wb(const void*, const void*)`
    - Description: Write barrier function that must be used after pointer writes to heap-allocated objects – the value of the field being written must also point to a heap-allocated object.

      If a generational collector is used, it may check whether the two function arguments are in different GC generations (i.e. if the first argument points to an old object and the second argument points to a young object), and if so, call the write barrier slow-path.
    - Required Annotations: `STATIC_INLINE`, `JL_NOTSAFEPOINT`.
- `void jl_gc_multi_wb(const void*, const void*)`

    - Description: Write-barrier function that must be used after copying multiple fields of an object into another. It should be semantically equivalent to triggering multiple write barriers – one per field of the object being copied, but may be special-cased for performance reasons.
    - Required Annotations: `STATIC_INLINE`, `JL_NOTSAFEPOINT`.

Allocation Fast-Path & Codegen Write-Barriers

Code to emit write-barriers into Julia's JITted code is currently located in two files in JuliaLang's repository:

- `src/llvm-late-gc-lowering.cpp`
- `src/llvm-final-gc-lowering.cpp`

More specifically, the emission of the write-barrier fast-paths (i.e., checking the bit patterns in object headers to detect whether there is an edge from an old to a young object in the object graph) happens in `LateLowerGCFrame::CleanupWriteBarriers` in `src/llvm-late-gc-lowering.cpp`. The lowering of write-barrier slow-paths (i.e., enqueuing an object into a remembered set), lowering of allocation functions, lowering of safe-point page reads, etc., all occur in the `FinalLowerGC` pass in `src/llvm-final-gc-lowering.cpp`.

We expect that third-party garbage collectors will implement both the `LateLowerGCFrame::CleanupWriteBarriers` function (but not the full `LateLowerGCFrame` LLVM pass) and the full `FinalLowerGC` LLVM pass.

This will enable customization of the write-barrier fast-paths and the lowering of allocation functions to possibly use custom allocation fast-paths in bump-allocators (as opposed to calling into the runtime to allocate).

Both `LateLowerGCFrame::CleanupWriteBarriers` and `struct FinalLowerGC` should be defined in the GC interface header.

Proposed File Organization

We propose the following changes to Julia's source tree:

- Introduction of a header file `gc-interface.h`, containing declarations of the functions described above. We also expect to move any other existing declarations into `gc-interface.h` so that these functions are declared exactly once. Additionally, we expect the definition of data structures that hold GC metrics (e.g., Julia's `GC_Num`) to reside in this file.
- Creation of header files `gc-stock-tls.h` and `gc-stock-runtime-barriers.h`, containing the definition of the GC thread-local storage used by Julia, as well as definitions of the write barriers used by the Julia runtime (these need to be in a header file so that they can be inlined).
- Creation of files `gc-common.h` and `gc-common.c`, containing the declaration and implementation of GC callbacks, finalizers, and other features we expect to be shared between MMTk and Julia's stock GC.
- Renaming of `gc.h` and `gc.c` to `gc-stock.h` and `gc-stock.c`.

For performance reasons, we will declare the runtime write-barriers as `STATIC_INLINE` in a header file. The pattern we will follow to ensure that we inline these short functions in the runtime is to conditionally `#include` in `src/julia.h` a header containing the write-barrier implementation of Julia's stock GC or the implementation of the third-party GC.

```
#ifndef THIRD_PARTY_GC
#include "gc-stock-runtime-barriers.h”
#else
#include <include/gc-runtime-barriers.h>
#endif
```

Figure 3: Conditional inclusion of the runtime barriers in `src/julia.h`.

This requires us to add the `include` directory from the MMTk folder into our include path if the third-party GC is enabled. Additionally, we need to ensure that MMTk has a `gc-runtime-barriers.h` header with the runtime write-barrier implementations in the `include` directory.

This conditional preprocessor flag pattern will also be used for conditional definition of the GC TLS, as outlined in section "Prelude: GC TLS for Stock Julia and for Third-Party GCs".

```
#ifndef THIRD_PARTY_GC
// contains the definition of the GC TLS for the stock GC
#include "gc-stock-tls.h”
#else
// contains the definition of the GC TLS for the third-party GC
#include <include/gc-tls.h>
#endif
```

Figure 4: Conditional inclusion of the GC TLS in `src/julia_threads.h`.

MMTk Fork Management & Versioning

Currently, MMTk relies on two codebases for the implementation of their custom garbage collector into Julia: the MMTk Core repository (https://github.com/mmtk/mmtk-core), which contains mostly language-agnostic garbage collector and allocator implementations, and the MMTk-Julia Binding (https://github.com/mmtk/mmtk-julia/tree/master), which contains garbage collector code that interfaces with the Julia runtime and is dependent on the layout of Julia objects (e.g., code to trace objects in the mark phase).

We expect the fork management and versioning to closely follow the process currently implemented for other dependencies. More specifically, we expect:

- To create an MMTk-Julia Binding fork inside the JuliaLang organization.
- To create a version file `mmtkjuliabinding.version` in the `/deps` directory of JuliaLang.

These versioning files will specify a branch and commit in the forks of MMTk inside the JuliaLang organization that will be used to build MMTk and the binding from source when the user specifies in their `Make.user` that they wish to use MMTk (see the "Build System Changes" section below).

Similarly to what the developers who maintain Julia's fork of LLVM do, we expect that the MMTk maintainers will be creating branches and opening PRs to the forks of MMTk inside JuliaLang, and also occasionally updating this fork to reflect upstream changes in MMTk Core.

An update of the MMTk version used in Julia will be marked by a change in the version files mentioned above (and also by changes in Julia's source tree itself required by the MMTk update).

Build System Changes

The second portion of this work will involve implementing build-system changes so that early adopters in the community can build Julia with MMTk by simply cloning the MMTk repository, adding `MMTK_GC=1` to their `Make.user`, and then building Julia from source with this flag enabled.

Building Julia with MMTk

We expect that a developer wishing to build MMTk with Julia to first:

- Clone Julia itself.
- Set `MMTK_GC=1` in `Make.user`.

We plan to create a rule in Julia's Makefile to clone MMTk if `MMTK_GC=1` is set in `Make.user`, and then invoke a build script that lives in the binding, which is responsible for building the binary for the alternative GC implementation. In the longer term, we should use BinaryBuilder to download a precompiled binary for MMTk, so that you don't need a Rust compiler on your computer.

As a result of building MMTk from source, if `MMTK_GC` is set, we will be compiling the alternative GC implementation into a `gc.o` object file, which will then be linked into Julia.

Testing

For testing purposes, we plan to move the benchmarks in the `serial` and `multithreaded` categories (which should take less than 5 minutes to run) from `GCBenchmarks` (https://github.com/JuliaCI/GCBenchmarks/tree/main) into a GC stress test in JuliaLang.

We aim to create a CI job in JuliaLang to run this GC stress test on Linux x86_64, using both the stock GC implementation (to ensure the stability of the current de facto GC implementation) and MMTk (to ensure that code changes in other parts of the compiler or runtime don't introduce code-rot in the Julia version running with an alternative GC implementation).

Ensuring that the GC stress test passes on the Julia version running on MMTk will be the responsibility of the JuliaLang PR authors. In a sense, the introduction of MMTk during this testing period will require the community to maintain two GC implementations.

In the first weeks of the integration, we expect significant support from the MMTk team in this maintenance process. This support will be augmented by a document describing how to contribute to and debug the MMTk binding.

While the JuliaLang PR authors will be responsible for ensuring that the current version of MMTk, as defined by the `mmtkjuliabinding.version` file, passes the GC stress tests, the MMTk team will be responsible for ensuring that PRs to JuliaLang, which merely bump the MMTk Core & Binding versions, are sufficiently stable when merged into Julia's master branch.

Appendix

References and further reading

- https://github.com/mmtk/mmtk-core
- https://github.com/mmtk/mmtk-julia
- https://github.com/JuliaCI/GCBenchmarks/tree/main