Precise Stack Scanning in C++

Attention: Can be shared Google-externally

This document describes an intern project with a research flavour where not all outcomes are clear yet. The project is about evaluating how Oilpan can be changed to be able to perform precise stack scanning using clang/llvm. V8 already has precise stack scanning using handles to keep track of on-heap pointers on the stack. A followup project is to evaluate the efficiency of that approach and whether it can be replaced.

Background and motivation

Oilpan is used to manage C++ memory in Blink. It is connected to V8 via unified heap. C++ code may keep references to objects from its native execution stack. While V8 and Oilpan try to finalize garbage collections through non-nested tasks, which are guaranteed to not have heap references on the stack, some garbage collections have to be finalized with a C++ stack due to timing reasons and resource limitations. This is ~11% of all Oilpan GCs (3.6% conservative + 9.3% unified heap where V8 finalizes via stack guard). Oilpan does not provide any garbage collection infrastructure to record pointers from the stack, meaning that it needs to conservatively walk the stack (word-by-word) and figure out whether references point into its heap. Upon finding such a reference, an object needs to be kept alive, i.e., it will be marked. This is called conservative stack scanning.

Conservative stack scanning comes with the following problems:

1. It is slow since all references on the stack have to be checked.
2. It potentially keeps more memory alive. E.g., consider a `double` or `intptr_t` (or multiple `int`s)value that looks like a pointer into the heap. Such a value would retain an object that may actually be unreachable. Also, external people have noticed that certain macros, e.g. VLOG, may leave uninitialized holes on the stack where old values may peek through.
3. A consequence of potentially misinterpreting primitive values such as doubles as an on-heap pointer is that Oilpan cannot run compaction on such garbage collections as the GC should not update double values. Precise stack scanning allows performing compaction on any GC.

Explorer: Using clang/llvm infrastructure

The idea is to leverage clang and llvm backend infrastructure to emit precise stack maps. This should work based on C++ types denoting that object are used on the garbage-collected heap of Oilpan. All interesting objects inherit from `GarbageCollected` or `GarbageCollectedFinalized` (with a few minor exceptions, see e.g. `HeapVector`), which can be used to figure out which pointers should be kept in a stack map.

Milestones

- **M1**: Explore using C++ types and clang/llvm to build pointer maps.
  - This can be a prototype that tries to emulate the environment in Blink (GCed base class) but can live completely outside of Blink and V8.
- **M2**: Integrate in Blink using already existing clang plugin infrastructure.
- **M3** (stretch): Productionize Blink prototype

- **M4** (optional): Investigate how the stack maps can be integrated in V8's handle system, which is `v8::internal::Handle` (internal) and `v8::Local` (API) level. This also requires checking build dependencies when integrating V8 into Blink.

Potential project impact

As of today it is not clear whether the approach is feasible (explorer in **M1**). In case the stack map works out, Oilpan is made fully precise, allowing compaction at any point in time and reducing floating garbage (**M2**, **M3**). Since Oilpan can work with conservative stack scanning, the precise approach is completely optional.

The project also serves as explorer to replace V8's handle mechanism, both internally and on V8 API level. This potentially has large performance impact on V8 and the web platform in general (**M4**).

Oilpan characteristics

- Code lives in `third_party/blink/renderer/platform/heap/`.
- GC entrypoint for full garbage collections: ThreadState::CollectGarbage

Most objects inherit from `GarbageCollected` (see here) or `GarbageCollectedFinalized`. Collection objects like HeapVector and friends use `IS_GARBAGE_COLLECTED_TYPE` macro to annotate GCed behavior.

Precise Stack Scanning with Clang Address Space Attributes

In Oilpan, most on-heap objects inherit from either a GarbageCollected or GarbageCollectedMixin base class. However, we can't populate GC stackmaps sections directly by checking for these bases in LLVM IR because of the possibility of Empty Base Class Elimination. To precisely identify which pointers need including in a stackmap at the IR level, we need an additional form of annotation for GC roots that we can be sure won't be optimised away.

Clang's address space annotations are a good fit here. It is supported directly in LLVM's recent(ish) GC API. Address space attributes are also used in OpenCL to specify that objects reside in distinct regions in memory, which allows them to be used for alias analysis. Using address spaces has two key benefits: (1) they are type qualifiers, which means the type system will help ensure that their usage is propagated correctly; (2) their semantics are decided by the implementor, which gives us a blank slate to use them for GC. The downside is that address space attributes are quite verbose and their pervasive use can make a codebase harder to read.

Why it is not possible to use Clang Plugins to add address space annotations to the AST

The original idea in Blink was to use a Clang plugin (similar to those in `blink_gc_plugins/`) to differentiate between on and off heap pointers and annotate them accordingly. The AST in Clang is generally considered to be immutable, but we had identified several undocumented APIs which do allow us to change the type of a node in-place.

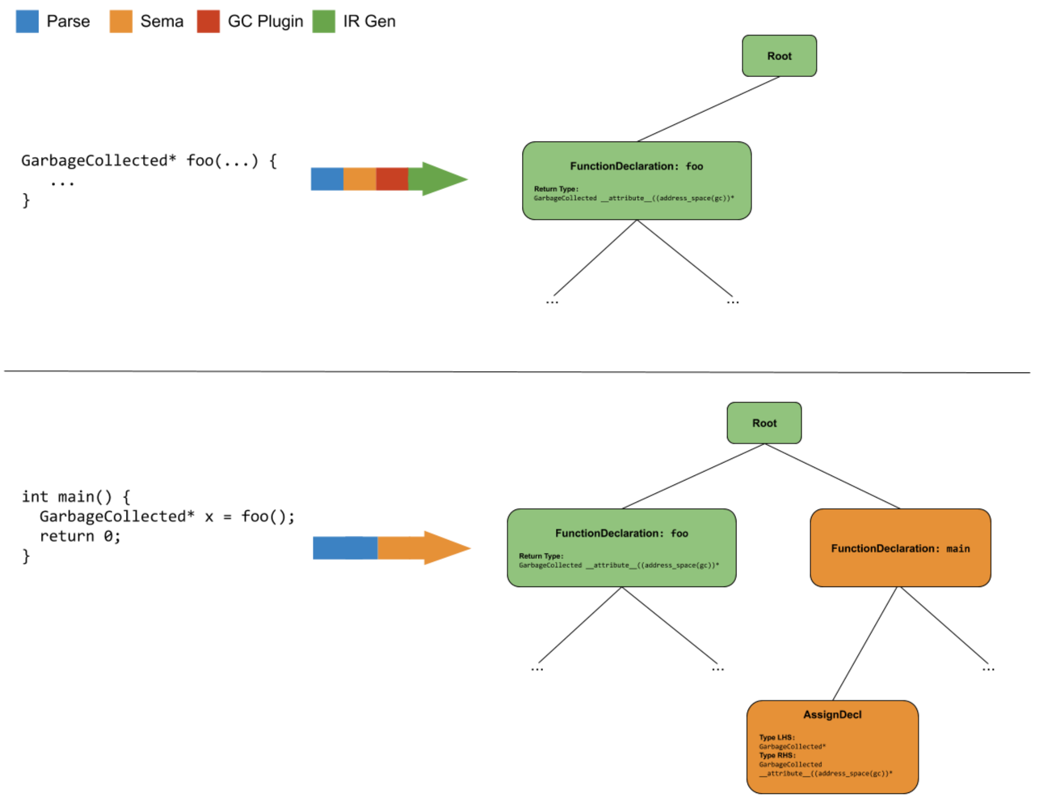
Unfortunately, this is a dead end due to Clang's non-obvious pipeline. By default, Clang plugins are run *after* the main AST action (which, in most cases, is IR generation). This can be mitigated by specifying a different action type to run before the main action.

There are two common entry points where Clang plugins regularly hook into in order to traverse the AST: HandleTranslationUnit and HandleTopLevelDecls. Most Clang plugins hook into the former, which allows them to walk a fully-formed, type checked AST. However, HandleTranslationUnit always happens after the main AST action (IR generation in our case) which makes it unsuitable for our use-case.

HandleTopLevelDecls is also unsuitable, but the problem is more subtle. Even when we set the Clang plugin to run before the main AST action, the pipeline for each top level declaration is as below:

Parse -> Typecheck -> Run Plugins -> Main Action (IR Gen)

This is problematic because for each declaration which is **not** defined at the top level, we are unable to mutate their AST nodes early enough to guarantee that the invariants of the type system hold. This is best explained with a diagram (many AST details elided for simplicity):



```
GarbageCollected* foo(...) {
    ...
}
```

```
int main() {
  GarbageCollected* x = foo();
  return 0;
}
```

Clang fails here when it reaches the line first line in `main()`. The type checker is run on the assign statement before we've had a chance to update the lvalue to be an address spaced `GarbageCollected` type (Which happens only after the entire top level decl is parsed). The result is a type error because the rvalue's type was updated in the previous top level declaration's pass.

It should be noted that the notion of using different address spaces to differentiate between pointer kinds is still a likely candidate for precise GC, however, more thought is needed to determine how to retrofit them to Blink's large codebase. Without forking Clang (or merging changes upstream) the following options are available, each with significant disadvantages.

Source-to-Source translation

This is the intended use-case for clang plugins where code changes are necessary. Instead of mutating the AST directly and then moving on to IR generation, the plugin outputs a new source file with the address space annotations inserted. This can be run once as a mass refactoring with the intention that programmers manually annotate address spaces in the future.
Pros: No effect on compile time. Simple to implement
Cons: Address space annotations are noisy and will clutter the codebase; more for the programmer to remember each time; mass refactor will likely touch every file in blink, and makes git blame / log tools harder to use.

Source-to-source as part of the build

Similar to above, but instead of running once as a mass refactor we include this as an additional step in the build.
Pros: No effect on the codebase; programmers can continue using blink as they were.
Cons: Duplication of work in parsing and type-checking the AST potentially causing a significant increase in build time. In essence we 2x the front-end phase of compilation.

Mutate the AST with non-type-qualifier annotations to use in LLVM

We can't operate directly on LLVM IR at present because we may have lost class hierarchy information once the AST is lowered. It might be possible to mutate the AST in-place, inserting annotations which survive lowering which, unlike address space annotations, are **not** part of the type system. An additional llvm opt pass could then run, transforming the address spaces of pointers which are tagged with these annotations.
Pros: Solves the duplication of work problem with S2S translation; custom annotations which are not typed will not break typing invariants on the AST.
Cons: The type system is actually very useful when adding address space qualifiers, it helps catch corner cases that have been missed and guarantees that their usage is propagated correctly. We lose a very useful soundness guarantee without it.

LLVM's Statepoint GC API

This API provides intrinsics and IR passes which allow for precise garbage collection in LLVM. In other words, GC roots can be precisely identified at program safepoints, making interaction with a moving garbage collector possible. It has been used successfully in Azul's Falcon JVM implementation (PDF), and the CoreCLR runtime.
It assumes the following has already been decided and / or generated by the front-end:

- A mechanism for determining where to place safepoints (e.g. across function boundaries, loop back-edges, both etc.)
- All managed and unmanaged pointers have been distinguished, and reside in separate, non-overlapping address spaces.
- The statepoint specific IR passes are run as late as possible in the IR phase ordering. This is because they can impede important optimisations such as inlining and dead code elimination.

For V8 and Blink, the statepoint API solves three problems:

- It provides support for parsing pointer location information into stack maps at given safepoints in the program.

- It generates stack maps and places them in a section in the final binary, allowing them to be queried by a collector at runtime.
- It encodes the semantics of a *potential relocation*[1] into the IR. In other words, variables which contain managed pointers are reassigned to fresh variables after they cross a safepoint boundary to account for the possibility that their value may have been relocated.[2]

Terminology

In GC literature, *safepoint* is a somewhat overloaded term: it can refer both to a point in a program where the GC can parse its machine state, and also the coordination required for threads to handshake such that a poll to the GC can be executed safely. The LLVM docs refer to the former exclusively with the term *statepoint*. For consistency, this document will do the same.

A pointer is considered *managed* if it points, either directly or transitively to a value on the GC-managed heap. Any other pointer is therefore considered *unmanaged*. These are also commonly referred to as on-heap and off-heap pointers respectively.

Proof of concept precise GC support in V8 with LLVM

This section will outline the bare-minimum requirements for sound precise stack scanning in V8 with the LLVM statepoint API. In this naive approach, there are clear performance pitfalls which until optimised, would not meet expectations for a production ready system. In later sections of this document, each performance concern is evaluated in depth, with potential optimisations explored. In some cases, there are areas with obvious performance wins can be achieved. The information in this section alone is not enough for production ready performance expectations.

Differentiating between managed and unmanaged pointers

This step is straightforward, as the current handle based GC abstraction ensures that there is a clear boundary in the V8 codebase between managed and unmanaged pointers. All on-heap objects must be accessed through managed pointers. In other words, no raw pointers to the GC heap are allowed. In using Handles, V8 is able to guarantee that references on the stack never point directly to a GC object, and that all roots can be precisely located by the collector by iterating Handle scopes.

In this proof of concept, we use an address-space of (1) to denote a managed pointer, and the default, elided zero address space for all other pointers (see explanation). With this approach, the (simplified) handle looks like this:

```
class Handle {
    Address* __attribute__((address_space(1)) address_;
}
```

[1] Even though in the common-case the GC will not be invoked at a safepoint poll, the only guarantee we have at compile-time is that it *might* have. As a result, the object could have potentially been relocated, and we need semantics to account for this.
[2] LLVM IR is in SSA form, so variable assignment is expected to happen only once. In order for GC relocation not to break this invariant, an assignment to a new variable is required.

This annotation makes up part of a type qualifier, and propagates through to the IR. Examples of the various stages of IR can be seen in the tests/out/ directory of the stack map artefact.

Identifying which functions need statepoints

In this proof of concept, statepoints exist only across a function call. Statepoints are not free: their corresponding stack map record will increase binary size; and they have the potential to inhibit llvm optimisations when lowering from IR to assembly. It is therefore desirable to make sure that they are used as sparingly as possible. The minimal number of statepointed call sites necessary which guarantees precise collection are any frames which a) contain managed pointers and b) potentially live on the stack during a collection.

For simplicity, this proof of concept overapproximates the number of statepoints needed with the following rule: any function which contains managed pointers is statepointed. There is scope for optimisation efforts here, but the work is non-trivial. See explanation here

Mitigating the effects on optimisation

There are concerns about the possible effects these changes may have on traditional compiler optimisations. This is roughly because:

- The gc statepoint intrinsics may prevent dead-code elimination and constant propagation
- Statepointed functions in the IR can prevent inlining
- If run before mem2reg or SROA, statepoint relocation is error prone and can prevent these important optimisations
- The introduction of new SSA variables for relocated values may have an adverse affect on register allocation.

The most important way to reduce interference on optimisation is to run the LLVM RewriteStatepointsForGC (RS4GC) pass, which introduces statepoint relocation sequences, as late as possible in the pass manager. In the stack_maps/ artefact, compilation is split and performed in stages because of a bug in Clang. The RS4GC pass is run after both clang++ -O2 -emit-llvm frontend, and opt -O2.

The inline problem

LLVM's RS4GC pass attempts to locate base pointers for potentially derived pointers. Derived pointers are not allowed in V8, however, in ToT LLVM there is no way to disable this check. In addition to the unnecessary work involved at compile time, this is problematic because of how LLVM performs this check.

For base pointer identification to work, the invariant that there are no inttoptr (or ptrtoint) casts visible to the RS4GC must be upheld. When this invariant is broken, RS4GC will trigger an assertion fail, leading to a CTE.

This becomes a problem with inlining because even if leaf functions which contain casting are deliberately excluded from the RS4GC pass (i.e. their function defs are not annotated with a gc strategy), they may be inlined into their caller before RS4GC is run.

In the future, it will probably be useful to feature gate base pointer identification so it can be switched off when not needed. This could be upstreamed into LLVM and used with newer rolls of Clang. However, in the meantime there are three potential workarounds:

1. Replace integer pointers with void*, this side-steps inttoptr casts which crash when observed by RS4GC.
2. Prevent functions containing casts from being inlined in the first pass, then, after RS4GC, run the inliner again on those specific functions. This can be achieved by making use of a custom "inline-later" annotation. Example in artefact.
3. Hide casting by making the function in which it occurs opaque. For example, compile it and link it as a separate translation unit. This only works if the function in question does *not* require statepointing. In addition, this can prevent inlining as well as other optimisations. Use this as a last resort.

Requirements needed to ship production ready version

Better handling of derived pointers

Initially it was assumed that we would not need mechanisms in place to deal with derived pointers, as V8 only allows *tidy pointers* to the header of an object. While this is true at the source level, this cannot be guaranteed after compiler optimisations. The compiler may introduce temporary, *untidy* inner pointers to object fields which require both tracking and possibly updating (in a moving collector). For example consider the following code:

```
A[i, j] = 10;
A[i, k] = 20;
```

After common subexpression elimination, this may become:

```
t = &A[i];
*(t + j * sizeof (int)) = 10;
*('c + k * sizeof (int)) = 20;
```

In this case, t is an untidy pointer as it points inside A at some offset i. If an untidy pointer crosses a statepoint, the collector needs to know how to convert it back to a tidy pointer. In order to guarantee soundness, two things need to happen:

- The tidy pointer's life must be extended such that it is considered live at the statepoint
- Both the tidy and untidy pointers are relocated after compaction.

The LLVM Statepoint architecture does attempt to find base pointers for each corresponding derived pointers, but has two main problems which may make its use in V8 difficult:

- It disallows inttoptr (and ptrtoint) casts in LLVM IR. At present, this will not work with the current Handle.address logic in V8 handles.
- It emits a base and derived pair for each root. This doubles the number of entries required in each stack map record.

The introduction of compiler generated untidy pointers are rare. Eliminating the need for duplicate stack map entries (and emitting more efficient stack maps in general) is the easier of the two problems. One approach which doesn't require performing liveness analysis on the CFG is to is to consider making handles relaxed atomics, which may be enough to prevent the optimiser from interfering with base pointers. This however does come with obvious negative performance implications.

A proposal for a more efficient stack map format

The current stack map format provided by LLVM is inefficient. Its memory usage is unnecessarily large, and it needs to be parsed into a table which can be queried at runtime by the collector.

```
Header {
  uint8  : Stack Map Version (current version is 3)
  uint8  : Reserved (expected to be 0)
  uint16 : Reserved (expected to be 0)
}
uint32 : NumFunctions
uint32 : NumConstants
uint32 : NumRecords
StkSizeRecord[NumFunctions] {
  uint64 : Function Address
  uint64 : Stack Size
  uint64 : Record Count
}
Constants[NumConstants] {
  uint64 : LargeConstant
}
StkMapRecord[NumRecords] {
  uint64 : PatchPoint ID
  uint32 : Instruction Offset
  uint16 : Reserved (record flags)
  uint16 : NumLocations
  Location[NumLocations] {
    uint8  : Register | Direct | Indirect | Constant | ConstantIndex
    uint8  : Reserved (expected to be 0)
    uint16 : Location Size
    uint16 : Dwarf RegNum
    uint16 : Reserved (expected to be 0)
    int32  : Offset or SmallConstant
  }
  uint32 : Padding (only if required to align to 8 byte)
  uint16 : Padding
  uint16 : NumLiveOuts
  LiveOuts[NumLiveOuts]
    uint16 : Dwarf RegNum
    uint8  : Reserved
    uint8  : Size in Bytes
  }
  uint32 : Padding (only if required to align to 8 byte)
}
```

This stack map format requires a minimum of 40 bytes for each stack map record entry (i.e. each statepoint) assuming it contains a single location.

Reducing statepoint overapproximation

In the artefact, statepoint locations are overapproximated, with some functions having unnecessary stack map entries. A function requires a corresponding stack map if its frame contains a managed pointer(s) and:

- It calls an external function (either directly or transitively)
- It calls a function (either directly or transitively) which is known to trigger a collection

The problem here is that it's unknown whether an externally linked function, `foo` has a control flow path which triggers a collection. The only guarantee is that it *might*, hence we must conservatively include a stackmap entry for all functions which may exist on the stack when `foo` is called.

If we could statically the traverse the call graph for foo's translation unit, and determine that its path never triggered the collector, then we could remove stack map entries for functions in TUs which call `foo` externally. This might be possible with the following:

- An LTO which may allow us to perform a pass over the whole executable, removing stack maps which are unnecessary. This could be too late, however, as certain compiler optimisations (which could have been unnecessarily impinged by overuse of stack maps) will have already taken place.
- A fixed point analysis similar to gcmole which traverses over many different translation units and uses files to record state between them.

Fixing the inefficient and clunky build process

The ideal scenario is that all steps required to build V8 / Blink with stack map support can be managed fully using the clang++ binary. This is not currently possible as the GCStrategy library (used during stack map generation) is only instantiated when used with opt and llc. This doesn't affect build times too much, but it does increase the number of steps required and can make it harder to understand.

Compiler Gating

As this uses an LLVM specific API, it will need to be conditionally included when used only with the Clang / LLVM toolchain. This also means that if the intention is for this prototype to be used to eventually remove the need for HandleScopes, either gcc support will need to be dropped, or they need to exist and be used in parallel for systems that are not compiled with LLVM.

Stack walking

In the proof of concept, stack walking is done by jumping into caller frames through the frame pointer. In production this can't be relied on: calling conventions may differ between platforms; and optimisations exist which use the frame pointer as a general purpose register. Walking the stack with stack maps in such cases requires plugging into the existing stack traversal mechanisms used in V8 and Blink.

Porting to architecture other than x64

At present, the LLVM statepoint lowering pass has only been implemented for x86_64. Work will be needed to port this to additional architectures. Although tedious, this work is purely mechanical. See https://llvm.org/doxygen/StatepointLowering_8cpp_source.html.