Internalizing strings in OffThreadFactory

2019-12-04, leszeks@

LGTMs

Name	Write (not) LGTM in this row
verwaest@	solution 3 lgtm
ulan@	solution 3 lgtm
hpayer@	solution 3 lgtm
rmcilroy@	solution 3 lgtm
<please add yourself>	

Update (2019-12-10): It looks like the simplest, safest approach is Solution 3, iterating the heap after finalization. I'll also just allocate strings off-heap, hoping that large strings stay unique and small strings are small enough that the fragmentation cost is outweighed by the off-thread allocation gain.

The Problem

Currently the finalization stage of compilation takes the AST strings created during parsing, and internalizes them as part of allocating a Heap String. This means that the allocation and internalization are tied together.

When an OffThreadFactory wants to allocate a Heap String from an AST string, it can't directly access the string table, as this could be mutated by the main thread. So, we have two problems

1. We need to somehow update the string table with our new strings, and
2. We might allocate Heap Strings which already exist in the string table, and have to be de-duplicated for correctness (entries in the string table **must** be unique).

These operations (update and de-duplicate) have to be somehow synchronised with the main thread. An ideal solution would

1. Minimise main thread time,
2. Minimise fragmentation (dead objects in the off-thread pages)

Side problem: when to allocate Heap Strings

One decision to make is when Heap Strings should actually be allocated. This ends up being a direct trade-off of the two aims above (minimising main thread time and minimising fragmentation).

The aim of the entire project is, of course, to push allocation to the background thread, so it makes sense to allocate on the background thread. However, if these allocated strings end up being duplicates of strings in the string table, then they end up dead and causing fragmentation. So, an alternative is to delay Heap String allocation until synchronisation with the main thread. However, this necessarily means more main thread time, as the allocation is not free, and means we have to keep the AST Strings alive until main thread synchronisation.

On the one hand, allocation might be cheap, since there's no pointer chasing and most of the allocation is a simple memcpy. On the other hand, large strings (causing the largest fragmentation) are the most likely to be unique, so perhaps fragmentation is limited.

In the end, I think this aspect of the design does not matter too much, and we can pick whatever is simpler.

Solution 1: Record slots during setting

The original proposed solution (implied in previous design docs) is to record slots pointing to strings by adding a new Object field setter, something along the lines of:

```
class SharedFunctionInfo {
  void set_name(String string) {
    RawField(kNameOffset).store(string);
  }
  void set_name_and_record_slot(AstRawString* string, OffThreadFactory* factory) {
    RawField(kNameOffset).store(factory->placeholder_string());
    factory->RecordStringSlot(RawField(kNameOffset), string);
  }
}
```

```
class Factory {
  Handle<SharedFunctionInfo> NewSharedFunctionInfo(String name) {
    Handle<SharedFunctionInfo> shared = NewSharedFunctionInfo();
    // Use the normal setter.
    shared.set_name(name);
    return shared;
  }
}

class OffThreadFactory {
  SharedFunctionInfo NewSharedFunctionInfo(AstRawString* name) {
    SharedFunctionInfo shared = NewSharedFunctionInfo();
    // Use the special setter.
    shared.set_name_and_record_slot(name, this);
    return shared;
  }
  void RecordStringSlot(ObjectSlot slot, String value) {
    slots_.push_back({slot, value});
  }

  std::vector<std::pair<ObjectSlot, AstRawString*>> slots_;
}
```

Then, during merging, we could walk all these slots, internalize their strings, and update the slot value to point to the newly internalized string.

Pro: Very explicit about which slots are updated, lowest performance impact since we rely on the programmer to “get it right”

Con: We rely on the programmer to “get it right”, which means that they can get it wrong -- in particular, they might accidentally use the wrong setter and miss a slot. It’s annoying to deal with setters that either set a String or a different type (e.g. setting undefined on a string field). We have to weave through the fact that we have to record strings through a *lot* of code, requiring creative solutions like CRTP (see the factory implementation doc).

I believe that there *are* checks we could add to guarantee correct behaviour, but they would have to be runtime checks (e.g. on normal setters, forbid assigning non-read-only strings to fields in off-thread objects) so we might miss them on uncommon paths, and this doesn’t resolve the other cons.

Solution 2: Live with ThinStrings

We already support post-hoc internalization/de-duplication of existing strings, by converting them in-place into ThinStrings. So, we could record all strings allocated, by an allocation bottleneck in the factory, and on merging with the main thread, internalize all the strings, leaving ThinStrings in their place.

The problem is, ThinStrings have never yet been used in compilation artifacts, so there *might* be code that relies on all Strings in compilation artifacts already being internalized SeqStrings. If these correctness issues were fixed, there would still be a performance issue, as ThinStrings add a non-zero performance overhead to string operations. Some of this performance could be recovered by removing ThinStrings where possible in the runtime, and some could be recovered by path compressing them away in the GC (e.g. as a post-process step after marking), but this is additional complexity.

Pro: Use existing methods

Con: Possible (hard to prevent) correctness issues, runtime performance loss if implemented without additional complexity

Solution 3: Iterate the off-thread heap for String slots

We could, instead of recording string slots during write, instead walk the off-thread pages looking for string slots, and then perform much the same kind of internalization/update on main-thread merge as with Solution 1.

Pro: No need for special setters, lowest main-thread performance impact.

Con: Possibly large off-thread performance impact, from having to walk kilobytes of pointers (may not be so bad since we'll be able to skip over bytecode and string data).

Detecting string slots is trivial if we allocate Heap Strings off-thread. If we wanted to instead delay allocation, we'd need to represent AST string pointers in slots somehow. This could be done with an AST string table + indices + a reserved area on the heap, or it could be done by pointing to some foreign-pointer-like object (though this object would die after merging, so it would increase fragmentation a bit).

Solution 4: Record slots during setting with a write-barrier

Similar to Solution 1, we could record slots during setting as a "write barrier" rather than a function overload. Setters would check if a) the slot being set is in an off-thread page, and b) if the value being set is a non-read-only string. For this case, they would (somehow) get access to the off-thread factory, and record the string slot there.

Pro: No need for special setters or for heap iteration

Con: All setters now have an extra write barrier branch (maybe, depends on how we deal with the other write barriers), there's no obvious clean way to get the off-thread factory in the setter without passing it through as a parameter (maybe an extra slot on the page header?). We probably end up doing as many checks as a page walk in the end, since we have to check so many pointers.

One option for removing the latter con is to pass wrappers around AST strings to the setter, and having either that wrapper point to the factory (larger object so larger fragmentation) or store the set of slots on AST strings directly rather than the factory (larger AST string, slot clearing becomes arguably harder).

Non-solution: Access the string table synchronously

Above I refer to "synchronising with the main thread". In the general OffThreadFactory design, this would normally mean waiting until the "finalization" stage, where the off-thread pages are merged back into the heap. But, we could instead synchronise access to the string table, to make off-thread string allocation de-duplicate straight away.

TL;DR: Sounds good, doesn't work.

There are two basic invariants that this would break:

1. If we find a duplicate string already: off-thread objects shouldn't point into the main Heap. The main Heap is movable, and having off-thread objects pointing into it would mean a) walking them as part of marking (to record slots), b) updating those slot values, and c) adding write barriers to all these writes which *also* requires synchronisation with the main thread.
2. If we don't find a duplicate, and have to add one: the main thread shouldn't point to off-thread objects. If it did, we'd have all sorts of mess with the marker and sweeper being able to walk the objects in these semi-detached pages.

Not to mention, adding entries to the string table can cause allocation (from resizing the table), which can cause GC, which can cause all sorts of other problems.

Possibly, *possibly* there could be a way of making this work, but it'd be hard to define the invariants in a sufficiently safe way.