Binaryen Object File Support?

azakai / kripken

Implementation

Supporting wasm object files means knowing how to read and write them, which includes differences like separate code sections for each function. The biggest challenge is **relocations**. Binaryen IR already represents various things in a relocation-friendly way, including:

- Calls and Table elements refer to the function by a Name.
- Global gets/sets refer to the global by a Name.

Where a Name is already used, we probably don't need to add anything. But are Names good enough? Two possible issues:

- Names should match wasm names in terms of whether null terminators are allowed, etc. Do linker names match that?
- Binaryen IR Names can overlap between types, that is, a Function might be called $foo and also a Global may have the same name. Is that possible with linker names? If not that's probably fine as Binaryen Names may be more flexible in a way that is never exercised. But if linker names have some other form of flexibility Binaryen Names disallow, we may have issues.

Aside from existing Names, the changes we do need to make are to extend the IR to allow expressing the additional information relocations need, so that we can parse them into our IR and then convert the IR to wasm when writing.

More specifically, **relocation information should be represented in IR form in Binaryen IR as much as possible**. However, some relocation information may be best kept in data on the side. Specifically, **when processing an object file we should not parse custom sections into an IR form**, and instead just rely on relocations into them for necessary updates.

In more detail, we should parse the code, data, etc. sections into Binaryen IR normally, and with new IR support for relocations. But we should not parse e.g. DWARF sections into an IR: we do that currently, using LLVM's DWARF-YAML code, because we have to when we run after wasm-ld. But the relocation info already encodes every reference to a symbol which has an unknown address or may move. So we can just update the relocations, as the linker does. Doing it that way is much simpler and avoids adding relocation support to the YAML IR. In summary, **we represent code, data, etc. sections in IR and rewrite them, but leave custom sections like DWARF in binary form and only update relocations into them**.

Binaryen IR changes for relocation support include the following:

- A new **Relocation IR node** which contains the symbol name (as a Name), and an int64 addend (int64 is forward compatible with wasm64). This would replace a **Const** node that has a relocation into it (unlike a Const, the optimizer would not be able to do

anything with such a node, of course). This supports `R_WASM_TABLE_INDEX_SLEB` / `R_WASM_MEMORY_ADDR_SLEB` relocations (which of the two would be specified in the relocation type).
    - Relocation IR nodes must be supported as segment initializer values (in addition to const and global.get).
    - Note that we don't need to store a relocation type here. A relocation section would need to say if a relocation is an LEB, SLEB, or int32, but the Relocation node replaces a Const (in Binaryen IR we know more than the linker does) so it's definitely an SLEB, and whether it's a table index or memory address is known from the symbol table.
- **Loads** (and stores and other nodes with offsets) need to support a relocation on their **offset immediates**, for `R_WASM_MEMORY_ADDR_LEB`. **Adding a Name** for the relocation (the existing offset can function as the addend) would increase the size of those nodes, which is unfortunate. Another downside is that existing passes that look at the offset must check if there is a relocation as well before making changes. Other options:
    - Don't use `R_WASM_MEMORY_ADDR_LEB`. That is, when parsing such a wasm file we can replace that relocation of the offset with an added Const on which we put a relocation. Offsets have different semantics than the Load's pointer input (offsets do not wrap) so we would need to emulate a non-wrapping load and trap if necessary (that computation would need to include the Load's constant offset as well). (A post-link optimization could turn that back into a normal offset.)
    - A new RelocatableLoad node, identical to Load but with a relocation, and that the optimizer ignores.
- Relocations into **linear memory** require something, for `R_WASM_TABLE_INDEX_I32` / `R_WASM_MEMORY_ADDR_I32`.
    - We probably don't want to change the vector of u8s into a vector of Names. Perhaps each Segment could have both an array of u8s and of Names + offsets (or the latter could be on the whole Memory).
    - Or, we could have a restriction that any relocatable value in memory should have its own segment, and then such a memory segment would have a relocation name and type instead of bytes. However, while that is LLVM's default, it is possible to have multiple symbols in the same segment. If we want to "canonicalize" to a single one per segment we'd need to "break up" segments, which seems more complicated.
- The Linking section itself, including symbol table etc., will need to be represented in IR. In IR form we can use Names instead of indexes, and just create indexes for writing, as usual.
- Importable and exportable things like Functions and Globals will need to have information on their visibility and other flags that things in the symbol table can have. This could either be added to each of those existing classes, or it could be on the symbol table on the side.
    - Adding to the existing classes would be friendlier to optimizations that add or remove functions etc. as there will be just one place to update (however these

optimizations are less important as doing LTO-like things on object files is not that important, see later).
    - Adding to the symbol table would avoid multiple classes having the same extra visibility etc. fields. In addition, visibility is relevant for everything in the symbol table, which includes not just Functions and Globals (and other wasm entities we already represent in IR), but also data in memory segments. Having all the linking metadata in the symbol table may be simpler.
- Multi-memory / multi-table will require loads, stores, and table operations to be able to relocate their index immediate, when that proposal arrives.

Questions:

- `R_WASM_SECTION_OFFSET_I32` indicates a byte offset in a section. Which sections can this apply to? If it's a section we modify, we need to track and possibly adjust it. For code, we already can track things (this was added for DWARF) but we may need to add more if necessary. For example, if something wants to track byte offsets in the type section, we don't track that currently.

Optimizer Changes

To begin with, it may be simplest to only do function optimization passes when working on a wasm object file (by disabling `addDefaultGlobalOptimization[Pre|Post]Passes`). That will ensure we do not change global things like adding or removing functions, changing data segments, changing globals, etc. For example, this would disable MemoryPacking (which otherwise would need to start to understand relocations), disable Inlining, etc.

In the long term we could support most of those. But it may not be necessary, as "LTO-like" optimizations that change global state are not a high priority: for example inlining would already be done by LLVM on object files, and also running binaryen's optimizer after link, in a normal LTO-like way, is where most new inlining opportunities would be found.