Add DWARF Support for *yaml2obj*

Proposal for GSoC-2020

Basic Information	**Name** Xing GUO **Email** higuoxing@gmail.com **Timezone** UTC+8
Education	**Southeast University**, Nanjing, China Master student, expected graduation date: June, 2022
Related Experience	- Learned DWARF debugging format. - Contributed to LLVM (mostly in Binary Utilities). [view my revisions]

Abstract

LLVM offers 2 useful YAML tools, *yaml2obj* and *obj2yaml*. The former one reads YAML files and emits object files, e.g., ELF, COFF and Mach-O. The latter one does the reverse, reads object files and emits YAML files. We use these tools to write unit tests for binary tools, e.g., *llvm-objdump*, *llvm-nm*, *llvm-readelf*, as YAML is easier to read and edit than raw assembly codes and pre-built binaries. More importantly, YAML keeps our tests code base maintainable. However, *yaml2obj* (ELF part) currently doesn't support generating DWARF sections very well (we have to hardcode the *Content* field of DWARF sections and it is not intuitive). This project aims to add DWARF support for *yaml2obj*, which will ease our pain crafting DWARF tests.
* Note: If it isn't pointed out specially, the object file we mention below refers to ELF object file.

>>> Updated Proposal <<<

Implementation Plan

Current Status of DWARF Support

Originally, *yaml2macho* first had DWARF support. After patch [llvm-svn: 288984] landed, *yaml2dwarf* was pulled out from *yaml2macho* as a generic library *DWARFYAML* aiming to support other object file formats. Unfortunately, up till now, only *yaml2macho* has DWARF support. Currently, *DWARFYAML* supports generating some commonly used sections, and lacking support for some new sections that were introduced in DWARFV5. See table shown below.

DWARF Sections (in alphabetic order) [2]	**Description** [1][2]	***DWARFYAML* Implementation**
`.debug_abbrev`	Abbreviations used in `.debug_info` section	Yes
`.debug_addr (V5)`	This table contains all addresses and constants that require link-time relocation, and items in the table can be referenced indirectly from the debugging information.	No
`.debug_aranges`	Lookup table for mapping addresses to compilation units	Yes
`.debug_frame/.eh_frame`	Call frame information	No
`.debug_info`	The core DWARF information section	Yes
`.debug_line`	Line number information	Yes
`.debug_line_str (V5)`	Contains strings for file names used in combination with the `.debug_line` section.	No
`.debug_loc` `(Deprecated in V5)`	Location lists used in `DW_AT_location` attributes	No
`.debug_loclists (V5)`	Location lists are used in place of location descriptions whenever the object whose location is being described can change location during its lifetime	No
`.debug_macinfo` `(Deprecated in V5)`	Macro information	No
`.debug_macro (V5)`	Macro information (introduced in V5 for replacing `.debug_macinfo`).	No
`.debug_names (V5)`	Contains the names for use in building an index section	No

`.debug_pubnames/` `.debug_gnu_pubnames` `(Deprecated in V5)`	Lookup table for mapping object and function names to compilation units	Yes
`.debug_pubtypes/` `.debug_gnu_pubtypes` `(Deprecated in V5)`	Loopup table for mapping type names to compilation units	Yes
`.debug_ranges` `(Deprecated in V5)`	Address ranges used in `DW_AT_ranges` attributes	No
`.debug_rnglists (V5)`	Same as `.debug_ranges` with several improvements	No
`.debug_str`	String table used in `.debug_info`	Yes
`.debug_str_offsets (V5)`	Contains offsets of strings in `.debug_str`	No
`.debug_sup (V5)`	This section exists in DWARF supplementary object files providing information for establishing relationships between debug sections (See [2] 7.3.6)	No
`.debug_types` `(Deprecated in V5)`	This section holds DWARF type definitions	No
`.debug_cu_index (V5)`	The compilation unit (CU) index section	No
`.debug_tu_index (V5)`	The type unit (TU) index section	No

Besides, in LLVM, we have GNU-like object file dumping tools, e.g., *objdump*, *readelf* … which are still lacking GNU compatible DWARF support (though we've already had *llvm-dwarfdump*), as shown below.

Tool Name	**LLVM**	**GNU**
objdump	`--dwarf[` `=frames` `]`	`--dwarf[` `=rawline,` `=decodedline,` `=info,` `=abbrev,` `=pubnames,` `=aranges,` `=macro,` `=frames,` `=frames-interp,` `=str,` `=loc,` `=Ranges,` `=pubtypes,` `=gdb_index,` `=trace_info,` `=trace_abbrev,`

		`=trace_aranges,=addr,`
		`=cu_index,     =links,`
		`=follow-links`
		`]`

readelf	`Do not provide`
	`--debug-dump`
		`--debug-dump[`
		`=rawline,      =decodedline,`
		`=info,         =abbrev,`
		`=pubnames,     =aranges,`
		`=macro,        =frames,`
		`=frames-interp,=str,`
		`=loc,          =Ranges,`
		`=pubtypes,     =gdb_index,`
		`=trace_info,   =trace_abbrev,`
		`=trace_aranges,=addr,`
		`=cu_index,     =links,`
		`=follow-links`
		`]`

Hence, it's good for us to implement DWARF support for *yaml2obj*, then we could write simple, maintainable unit tests and add more useful features for LLVM binary utilities.

Implementation Plan

- Add a new optional DWARF entry for ELF YAML file. We mainly focus on DWARFv5 sections, since they are new and useful at the moment. Some DWARF sections have been included in the *ObjectYAML* library, some are not. In addition to porting existing DWARF support to ELF, we will add some new sections to *ObjectYAML* that were introduced in DWARFv5, e.g., `.debug_str_offsets`, `.debug_addr`, `.debug_line_str`, `.debug_loclists` and `.debug_rnglists`.

```
--- !ELF
FileHeader:
  ...
Sections:
  ...
DWARF:
  debug_abbrev:   ...
  debug_aranges:  ...
  debug_info:     ...
  debug_line:     ...
```

```
debug_str:      ...
debug_str_offsets:
  Length: <Number:4-byte or 12-byte>
  Version: <Number:2-byte>
  Padding: <Number:2-byte, must be 0 (reserved)>
  Offsets:
   - <Number:4-byte or 8-byte>
debug_addr:
  Length:   <Number:4-byte or 12-byte>
  Version:  <Number:2-byte>
  AddrSize: <Number:1-byte>
  SegSize:  <Number:1-byte>
  Entries:
    - Addr: <Number:AddrSize>
    ...
debug_line_str:
  - <String>
  ...
debug_loclists:
  UnitLength:       <Number:4-byte or 12-byte>
  Version:          <Number:2-byte>
  AddrSize:         <Number:1-byte>
  SegSize:          <Number:1-byte>
  OffsetEntryCount: <Number:4-byte>
  Entries:
    - Name: <String:[DW_LLE_*]>
      Other fields are determined by Name
    ...
debug_rnglists:
  UnitLength:       <Number:4-byte or 12-byte>
  Version:          <Number:2-byte>
  AddrSize:         <Number:1-byte>
  SegSize:          <Number:1-byte>
  OffsetEntryCount: <Number:4-byte>
  Entries:
    - Name: <String:[DW_RLE_*]>
      Other fields are determined by Name
    ...
```

- Implement *ELFEmitter* to generate DWARF sections for ELF YAML files.
  - **Collect Debug Sections in YAML**
    ELFEmitter uses `std::vector<std::unique_ptr<Chunk>> Chunks` to "collect" section descriptors, then serialize "chunks" into object files. If we have DWARF entry in an ELF YAML file, we firstly initialize them as implicit sections and insert their section names as placeholders into `Doc.Chunks`.

    ```
    // llvm/lib/ObjectYAML/ELFEmitter.cpp
    template <ELFT>
    ELFState<ELFT>::ELFState(...) {
      ...
      if (Doc.DWARF) {
        ImplicitSections.insert(ImplicitSections.end(),
           {".debug_abbrev", other debug sections ...});
      }
      ...
      // iterate over implicit sections and
      // insert section into Doc.Chunks.
    }
    ```

  - **Serialize Chunks into Object File**
    Implicit section contents are serialized into object files in `initXXXSectionHeader(...)`, which determines section header fields, e.g., `sh_info`, `sh_entsize` …, then writes section contents to `raw_ostream`. See: `ELFState<ELFT>::initImplicitHeader(...)` in ELFEmitter.cpp. Just as what these helper functions do, DWARF sections can be serialized in the same way, e.g., `initDebugStrSectionHeader(...)` for initializing and serializing `.debug_str` section, `initDebugInfoSectionHeader(...)` for initializing and serializing `.debug_info` section. Besides, *DWARFYAML* provides several useful functions for generating DWARF section content, and we should reuse them in the helper functions, e.g.
    - `void DWARFYAML::EmitDebugAbbrev(raw_ostream&,`
      `const DWARFYAML::Data&)`
      for generating `.debug_abbrev` section
    - `void DWARFYAML::EmitDebugAranges(raw_ostream&,`
      `const DWARFYAML::Data&)`
      for generating `.debug_aranges` section
    - …

- **Test Functionality of Our Implementation**
  There are several phases during our implementation, so our unit tests vary during different periods. The first phase is that after we add the DWARF entry for ELF YAML file, we will have a test on emitting a warning when *yaml2obj* tries to read a YAML that includes the DWARF entry. The second phase is that, when we implement functionality for parts of DWARF sections, we will have tests on the sections generating and emitting warning when the user specifies unimplemented sections. The third phase is that when we finish implementing DWARF features for *yaml2obj*, we will be able to remove unimplemented features tests and tidy up discrete unit tests for DWARF sections.

Expected Results

- Enable *yaml2obj* to generate following DWARF sections for ELF YAML files and have sufficient unit tests for it.
  - `.debug_abbrev`
  - `.debug_addr`
  - `.debug_aranges`
  - `.debug_info`
  - `.debug_line`
  - `.debug_line_str`
  - `.debug_loclists`
  - `.debug_rnglists`
  - `.debug_str`
  - `.debug_str_offsets`
- If I have some time left after the first goal is accomplished, I would like to work on *obj2yaml*, as *obj2yaml* and *yaml2obj* should be able to read each other's output, convert between object files and YAML files "over and over again" and the content will not be changed (I'll try my best to accomplish this).
- This is *not* a GSoC goal, but I'm glad to implement missing DWARF support for *llvm-objdump* and *llvm-readelf* in the future (once we have DWARF support in *yaml2obj*), as this benefits people who are accustomed to GNU Binutils.

Timeline

Date	Event	Schedule
~ June. 2	Community Bonding Period	Finish my existing revisions, familiarize myself with the ObjectYAML code base, Contact with mentor(s) to discuss this project.

June. 2 - July. 4	Phase 1	***Week 1 (May. 31 - June. 6)*** Add DWARF entry for ELF YAML file. ***Week 2 (June. 7 - June. 13)*** Add support for `.debug_str` and `.debug_line_str` generating (Since `.debug_str` and `.debug_line_str` are basic raw string sections, and depend on no section). Start to work on `.debug_str_offsets`. ***Week 3 (June. 14 - June. 20)*** Add support for `.debug_aranges` generating (.debug_aranges doesn't depend on other debug sections). And start to work on `.debug_line`, since it involves many fields, and hard to do well. ***Week 4 (June 21 - June. 27)*** Add support for `.debug_abbrev` generating. ***Week 5 (June. 27 - July. 4)*** Add support for `.debug_line` generating (`.debug_line` contains much information, I would like to spend 2 weeks working on this).
June. 30 - July. 4	Phase 1 Evaluation	
July. 4 - July. 28	Phase 2	***Week 6 (July. 5 - July. 11)*** Add support for `.debug_loclists` generating. ***Week 7 (July. 12 - July. 18)*** Add support for `.debug_rnglists` generating. Start to work on `.debug_info`, which depends on many sections, and I want to leave enough time for design and implement this. ***Week 8 (July. 19 - July. 25).*** Add support for `.debug_addr` generating. ***Week 9 & 10 & 11 (July. 26 - Aug. 15)*** Add support for `.debug_info` generating.
July. 28 - Aug. 1	Phase 2 Evaluation	
Aug. 1 - Aug. 25	Phase 3	***Week 12 & 13 (Aug. 16 - Aug 29)*** I would like to reserve two weeks for unpredicted events. And if I finish the listed tasks, I will start to work on *obj2yaml*.
Aug. 25 - Sept. 1	Final	Review my work during summer.

	Evaluation	

Reference

[1] Debugging Using DWARF

[2] DWARF Debugging Information Format Version 5