Smaller, Split, DWARF5

Perftools Toronto 2018

Mark J. Wielaard

redhat.

TL;DR (Summary)

DWARF is big and expressive, we want to make it smaller (and even more expressive). Various work has been done (in gcc, elfutils, gdb), but more needs to be done (to more tools) to make this the default.

There can be some debate on what “this” is.

- Minimal DWARF5
- Plus debug-types
- Minus most relocations (plus indexes)
- Combined with split-dwarf
- Focus on compile/edit/debug and/or deployment

Not going to cover...

- Making it more expressive
- Alex Oliva did a nice presentation on “gOlogy” (the study of gcc -O* vs -g)
  - Using source loci in stmts and insns, var-tracking, VTA, SFN and LVU to mask effects of optimizations.
  - http://www.fsfla.org/~lxoliva/writeups/gOlogy/

    (“For reading fun”, it is huge!)

What is DWARF

From binary to source

But if that was all:

.debug_line mapping

addrs -> source/line-no

What is DWARF

Once you have source you might want to know…

- Which function are we in, what parameters does it have, which variables are in scope? What are the types of those?

  .debug_info (DIE - Debug Information Entries)

- Given those variables, what are their values?

  .debug_loc (location descriptors)

What is DWARF

- What code range does this function (or lexical scope) span?
  .debug_ranges

- How did I get here? What were the values of the variables in scope when this function was called?
  .debug_frame (.eh_frame) unwind information

- Now that I have the source, how does the following code snippet expand?
  .debug_macro

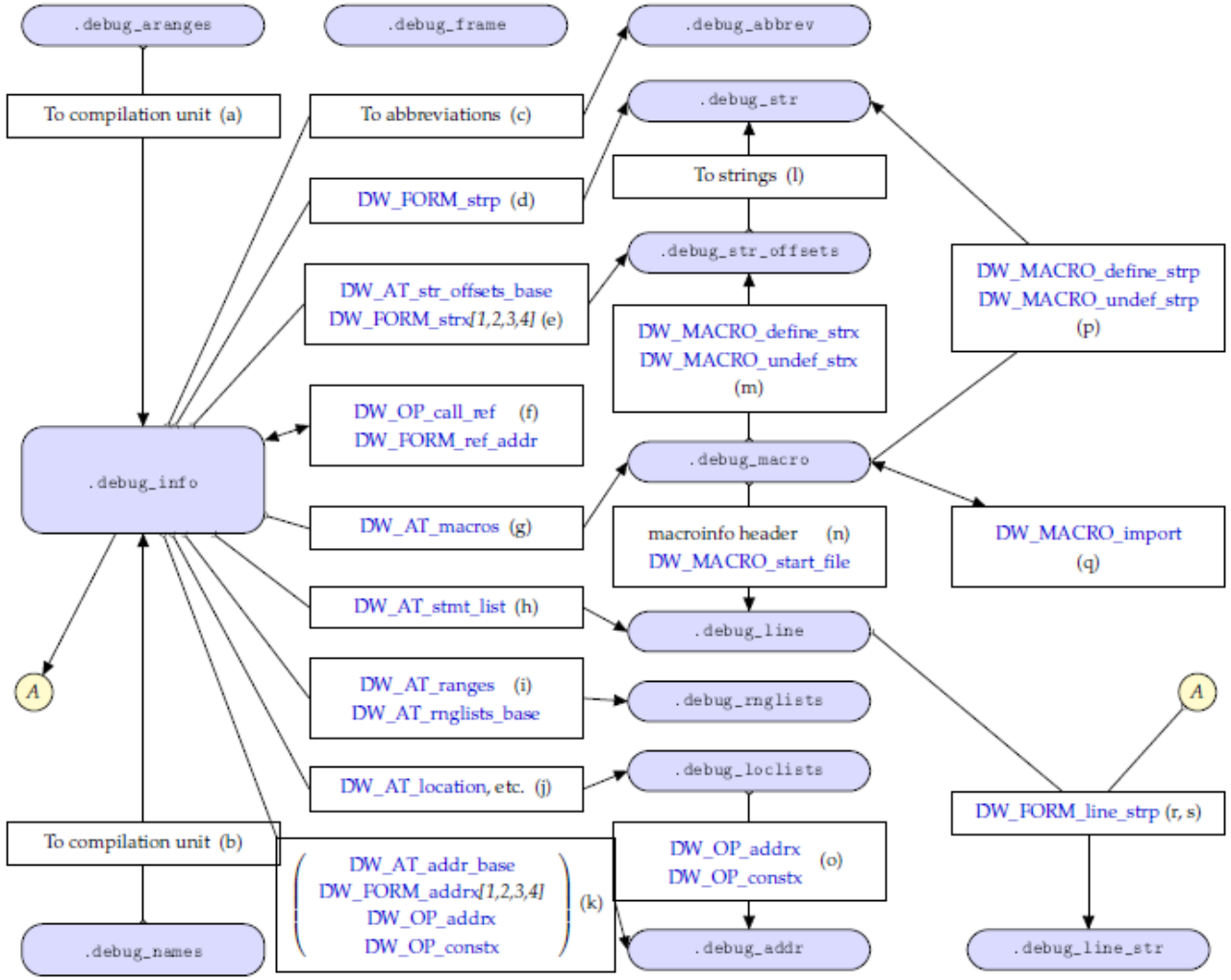


Figure B.1: Debug section relationships

DWARF5 additions

http://dwarfstd.org/Dwarf5Std.php

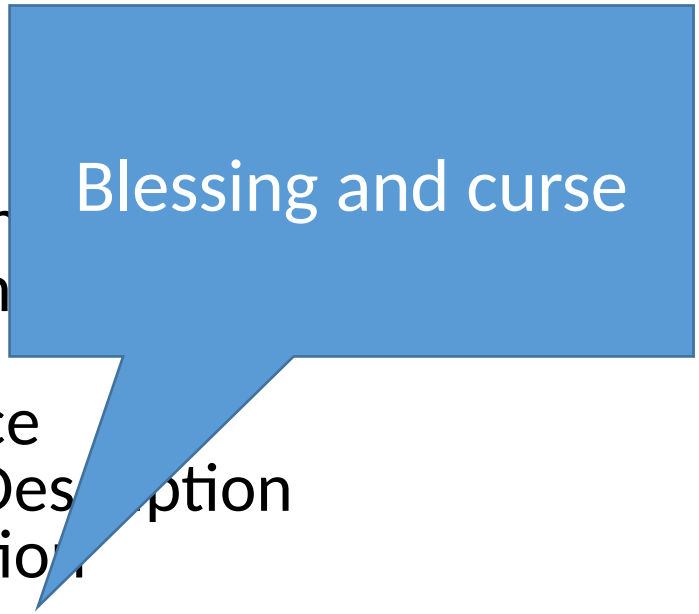
Too many to discuss here.
Focus on **new/smaller data representation**

DWARF standard design goals

- Language Independence
- Architecture Independence
- Operating System Independence
- Compact Data Representation
- Efficient Processing
- Implementation Independence
- Explicit Rather Than Implicit Description
- Avoid Duplication of Information
- Leverage Other Standards
- Limited Dependence on Tools
- Separate Description From Implementation
- Permissive Rather Than Prescriptive
- Vendor Extensibility

DWARF standard design goals

- Language Independence
- Architecture Independence
- Operating System Independen
- Compact Data Representation
- Efficient Processing
- Implementation Independence
- Explicit Rather Than Implicit Des ption
- Avoid Duplication of Informatio
- Leverage Other Standards
- **Limited Dependence on Tools**
- Separate Description From Implementation
- Permissive Rather Than Prescriptive
- Vendor Extensibility



Limited Dependence on Tools

DWARF data is designed so that it can be processed by commonly available assemblers, linkers, and other support programs, without requiring additional functionality specifically to support DWARF data.

What is needed to composite DWARF

- Being able to reference labels in data
- Being able to reference labels between data sections
- Being able to reference symbols
- Would be nice if assembler can produce leb128

And that is it!

With the above a DWARF producer can generate DWARF for an object that can be combined by a linker without any more special support. Just resolve the references/relocations and we are done!

Simple, but has issues

- Lots of duplication
  - Especially type descriptions in every object/CU.
- Relocations are “huge” (and too complicated)
  - Lots of intersection references
- Lots of data is just moved around
  - 50+% of an object file can be .debug data.
- No standards for deployment
  - Separate .debug files /usr/debug/src are “conventions”

Duplication Debug Types

- Skipping over this quickly, since feedback from GCC side was they really didn’t like this too much.
- This is a solution that relies on the tools/linker not knowing any DWARF

Duplication Debug Types

What if

- we had a "section group" or "linkonce section" where identical/duplicate sections would be merged/only one picked when combining object files?
- we define a hash/checksum over a type

Then

- we could put a type into such a data section with that hash/checksum as name

.debug_types

- Define a new way to refer to a type DIE (DW_FORM_ref_sig8) and the linker will make sure identically named ones will be de-duplicated.
- Does require a new DWARF unit header format that includes the sig8 and the offset into the DIE tree that identifies the type.
- DWARF4 put these into their own section (.debug_types), so they don't get mixed up with the "real" compile units (.debug_info).
- DWARF5 can be in .debug_info (identical unit headers).

Debug Types - Conclusion

Linkers all already have some kind of mechanism for this, so now we de-duplicate some information between DWARF in object files for "free".

But...

More complicated → two DIE data sections (.debug_info and .debug_types)

→ not enabled by default in GCC

- Maybe should be enabled by default for DWARF5?

(Still complicated inside split-dwarf .dwo files though.)

- Feedback at Cauldron was NO!

Relocations vs section refs

- Relocations on x86_64 (RELA) are big
  - 64bit address/offset
  - 64bit type and info
  - 64bit addend
- Plus 32bit/64bit for the actual ref in `.debug_info` section
- Minimum of 24 bytes!

DWARF5 references

- Base offset per CU (32/64 bits)
  - Not necessary in split-dwarf .dwo file.
- Add 32/64bit index at start of table (.debug_rnglists, .debug_loclists)
- Reference into this index table uleb128 a 1,2,3 or 4 byte ref.
- Ref is now just 5 to 8 bytes!

Replacing relocations

- A bit more work for the producer
  - Need to keep track of indexes/offsets
- Really does save lots of space and extra work for the linker
- GCC currently only does it when producing split-dwarf, but it really should do it always when generating DWARF5
  - **Feedback at Cauldron**: It is more subtle than that if you look at the final linked binaries. But yes, make it an option to always produce it.

split-dwarf

What if we could let the linker only deal with those parts of the DWARF data that needs relocations?

We already got rid of most inter-section references.

Why do we have most other relocations?

- Attributes referencing addresses/symbols
- Attributes referencing strings

A relocatable ‘skeleton’

- Only keep a skeleton DIE in the object file
  - Contains addr_base, low_pc, stmt_list
- Add an .debug_addr section that contains just addresses

  So instead of a direct relocatable reference we just index into this addr section.

- In the .dwo use a .debug_str_offsets as extra indirection layer

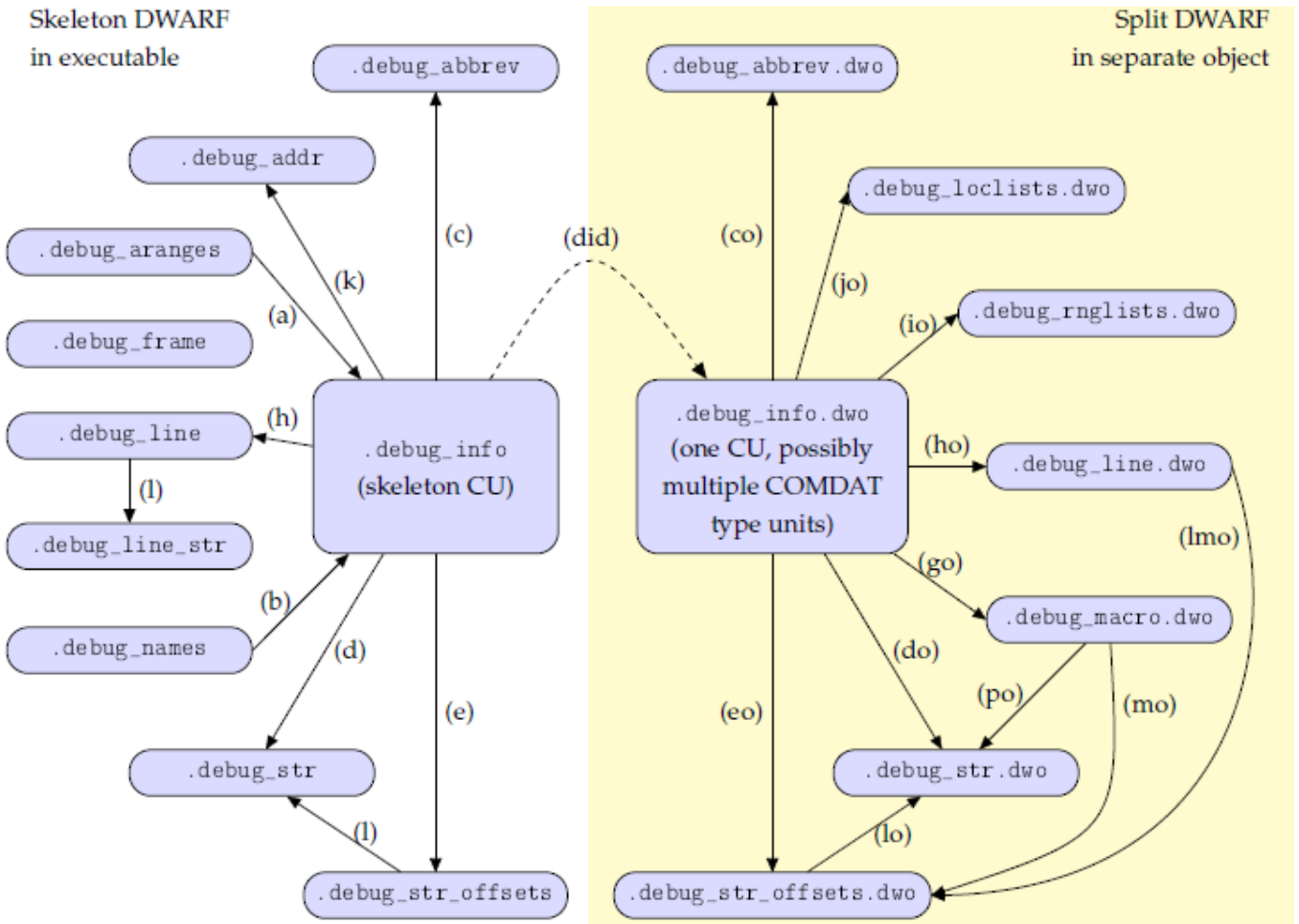


Figure B.2: Split DWARF section relationships

Conclusion split-dwarf

- Pretty awesome for your normal edit/compile/debug cycle. Linker “sees” a lot less data/relocations.
- Does require producer to keep track of address (indexes)
- We have lots of duplication again, but nothing that wouldn’t have already been in the .o object files in the first place (DWARF consumer need to be a bit smarter).
- Awful for distribution. There could be thousands (!) of .dwo files.

DWARF Package Files .dwp

Binutils dwp:

```
Usage: dwp [options] [file...]
  -e EXE, --exec EXE       Get list of dwo files from EXE
                           (defaults output to EXE.dwp)
```

Joins .dwo files into one .dwp file. Needs to know a little DWARF (at least read headers, signatures/dwo ids). Acts as mini linker joining string sections, updating .debug_str_offsets. Concatenating data sections and keeping track of offsets/sizes.

Works for DWARF4 + GNU DebugFission, but not yet DWARF5.

DWZ or DWARF Supplementary files .sup

- Since we are already writing tools that do need to know/transform DWARF data, why not go all the way!?!?
- DWZ de-duplicates not just between compile units, but even between debug files.
- Needs another set of reference forms
  `DW_FORM_GNU_ref_alt, DW_FORM_ref_supx, DW_FORM_GNU_strp_alt, DW_FORM_strp_sup`
- Such a tool (dwz) must know about DWARF, cannot be “simple” linker.

What did we learn from “producer side”?

- GCC is really good, has lots of infrastructure, other producers not so much…
- Standard is very focused on “simple” edit/compile/debug cycle solutions
  - Separate .debug files for example isn’t really standardized. Causes the “how to match up my objects/debug/source” issues.
- Post-processing (dwp, debugedit, dwz) is getting more important for production use (not “simple”).

Consumer support

- GDB debug types, split-dwarf or DWARF5
  - But not DWARF5 split-dwarf yet
    - Not completely GDB’s fault, I have patches for GCC 9, need backporting to GCC 8.x
- Elfutils debug types, split-dwarf and DWARF5
  - But not debug types and split-dwarf together
  - No dwp support yet

Distro/deployment tools

- Binutils DWP only does DWARF4 split-dwarf
  - Write an eu-dwp that does DWARF5?
- DWZ only does DWARF4 (no debug types, no split dwarf)
  - Make it work against .dwp files?
- RPM debugedit only does DWARF4 (no split-dwarf)
  - There is another copy inside flatpak!
  - The ugly “where is the source” problem

Other Consumers

- Systemtap and libabigail
  - Both need to adapt new elfutils DIE interfaces
- Valgrind

  - Rewrite DWARF reader based on (external) elfutils tool