# DWARF 5 and GNU extensions

New ways go from binary to source

Mark J. Wielaard

# Who am I

Mark J. Wielaard
Red Hat
perftools
valgrind, elfutils, systemtap, ...

redhat®

# 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 tree)

- 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 snippet expand?
  **.debug_macro**

# 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

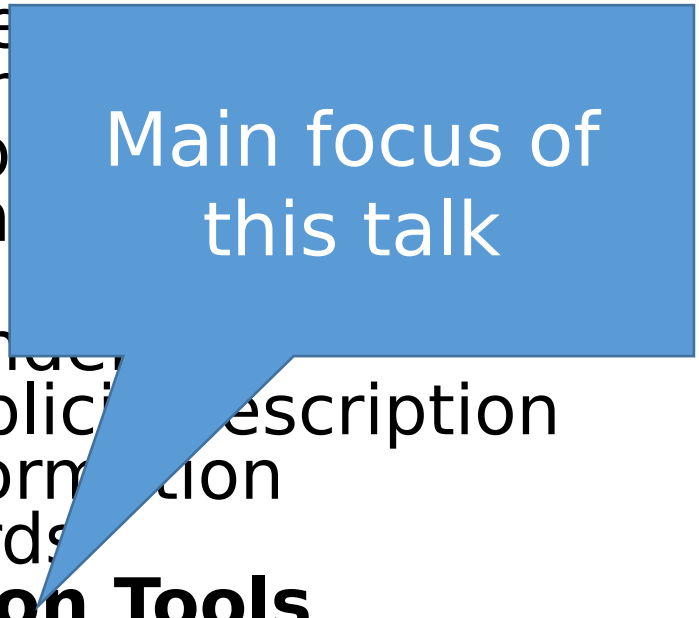
# DWARF standard design goals

- Language Indepen
- Architecture Indep
- Operating System
- Compact Data Rep
- Efficient Processin
- Implementation In
- Explicit Rather Tha
- Avoid Duplication
- Leverage Other Stand
- Limited Dependence Tools
- Separate Descriptio From Implementation
- Permissive Rather Than Prescriptive
- **Vendor Extensibility**

This means we can often try out stuff through a GNU Vendor Extension and then propose it for the next standard.

# DWARF standard design goals

- Language Independence
- Architecture Independen
- Operating System Indep
- Compact Data Represen
- Efficient Processing
- Implementation Indepen
- Explicit Rather Than Implici escription
- Avoid Duplication of Inform tion
- Leverage Other Standard
- **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.

There are some clever GNU extensions added to DWARF5 to counter some of these limitations.

# DWARF5 additions

http://dwarfstd.org/Dwarf5Std.php

Too many to discuss here.

Focus on **new data representation**

# 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 (relocations)
- 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.

# Example

- 2 source files, 1 header with data structure

- Parts of object file 1
- Parts of object file 2
- Combined
  - > just concatenate and resolve symbol relocations, inter-section references

# Example - conclusion

So, no really special magic needed to combine DWARF from separate object files.

But

- look at that repeated type
- and that are a lot of references/relocations

# .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?  (ELF Comdat sections)
- 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.
- So we put these into their own section, so they don't get mixed up with the "real" compile units in .debug_info.

# Example

Same example

Note: we only have one type DIE

# Example - 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

# GNU DebugFission or .dwo

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

Why do we have those relocations?

- Attributes referencing strings
- Attributes referencing addresses/symbols
- Attributes using inter-section references

# Add an indirection layer

- Add an .debug_addr section that contains just addresses
- Add a .debug_str_offsets section

  So instead of a direct relocatable reference we just index into these sections.
- Reduce the number of inter-section references to 1 by adding an index at the start (.e.g. for .debug_loclists and .debug_rnglists).

  Add “base” attributes DW_AT_str_offsets_base, DW_AT_addr_base, DW_AT_loclists_base, etc.

  And just use indexes against those.

# Example split-dwarf

Again the same example. But now with -gsplit-dwarf creating a skeleton unit in the main .o (with everything relocatable) and a .dwo file with everything else.

# Conclusion split-dwarf

- Pretty awesome for your normal edit/compile/debug cycle.
- We have lots of duplication again, but nothing that would 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.

EXAMPLE

# 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`

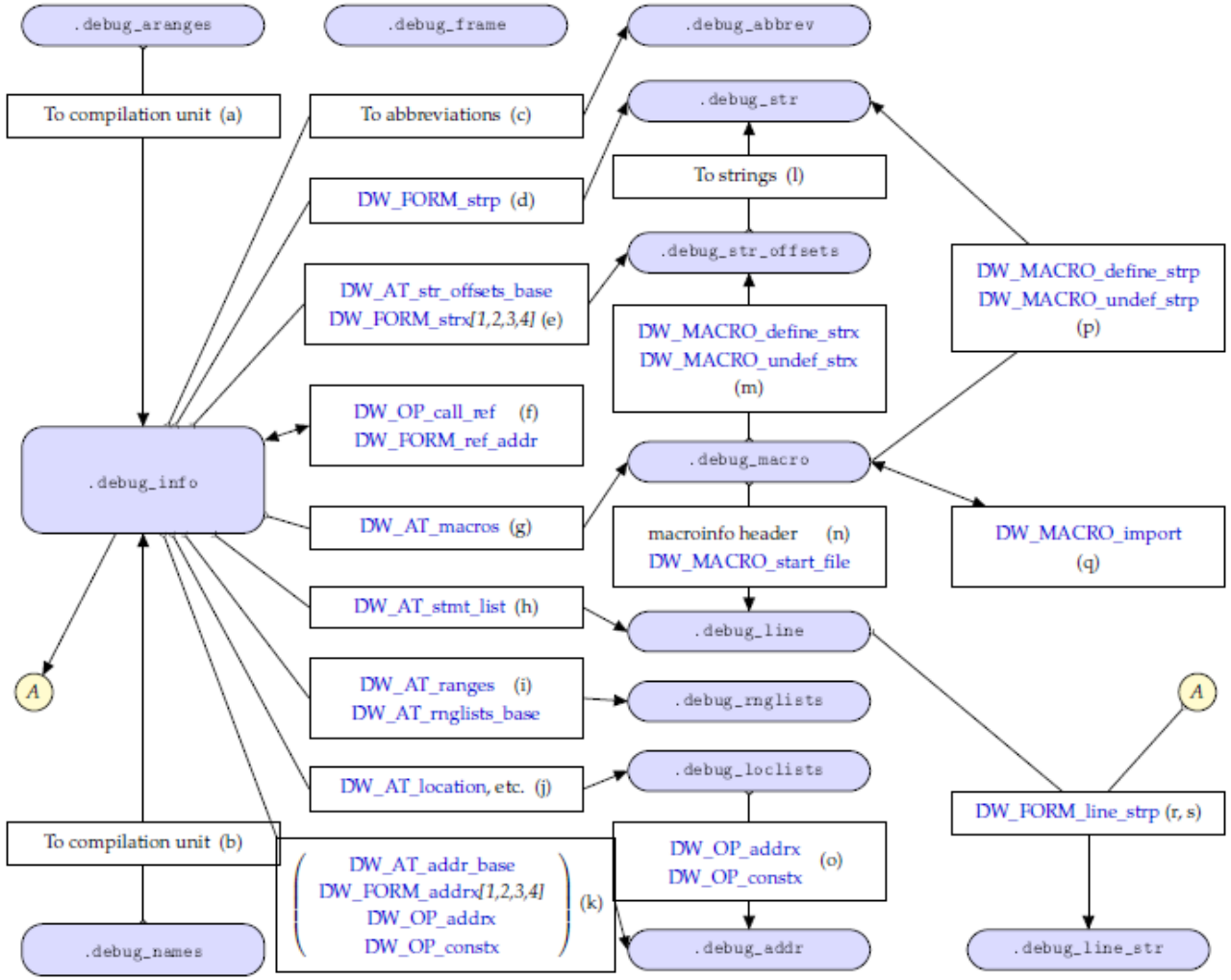



Figure B.1: Debug section relationships

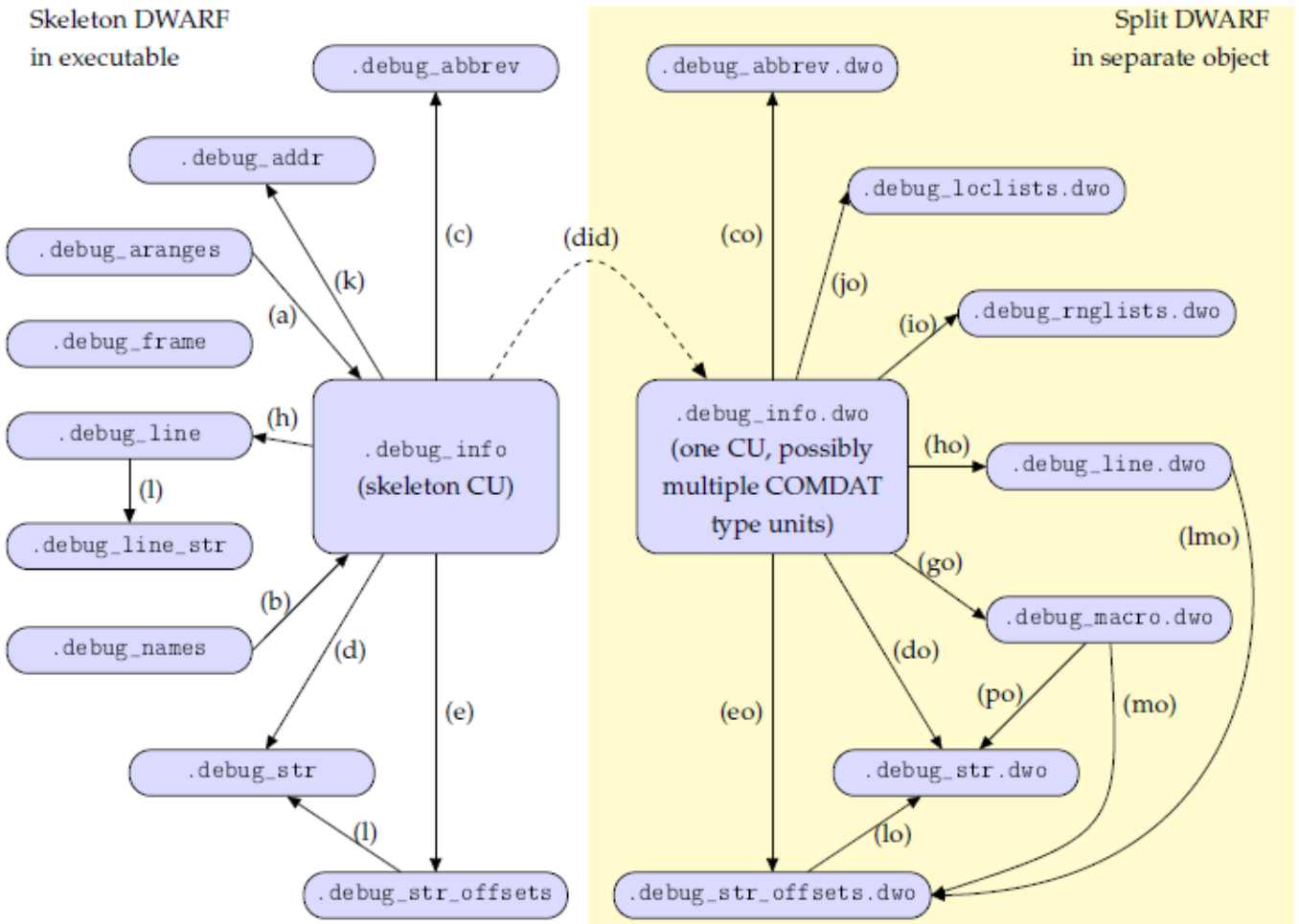



Figure B.2: Split DWARF section relationships

# Advertisement

If writing a DWARF consumer you might want to use a library.

Why not try elfutils libdw?

**ELFUTILS**