Dead Debug Data Elimination Using Fragmented DWARF

James Henderson
SN Systems (Sony Interactive Entertainment)
LLVM Developers’ Meeting 2020
The Problem

ELF object (approximate representation)

Headers etc

FUNC1()
FUNC2()
FUNC3()

.text

.DWARF

.debug_info
The Problem

References

Headers etc

.func1()
.func2()
.func3()

.text

.DWARF

.debug_info
The Problem

Clang -ffunction-sections ("section per function")
The Problem

Also happens in other situations, e.g. COMDATs

References

```
Headers etc

func1()
func2()
func3()

.text.func1
.text.func2
.text.func3

.DWARF

.debug_info

clang -ffunction-sections
```
The Problem

- Headers etc
- func2()
- .text.func2
- .debug_info
- Removed functions
- References
- ld.lld --gc-sections
The Problem

References from DWARF do not inhibit section removal.

No DWARF removed: leaves dead references.
Fragmenting DWARF (example)

- Example (simplified) .debug_info tree.

- Contains 3 functions (“subprograms”).

```plaintext
DW_TAG_compile_unit
  DW_TAG_subprogram
    DW_TAG_formal_parameter
DW_TAG_namespace
  DW_TAG_subprogram
DW_TAG_subprogram
  DW_TAG_formal_parameter
DW_TAG_base_type
```
Fragmenting DWARF (example)

- Fragment by splitting into “generic” bits and parts for specific functions (and variables).
- Result is 6 separate `.debug_info` sections.
- `.debug_info` sections for functions linked to corresponding `.text` sections.
If linker discards .text section, it also discards associated .debug_info section.

- DW_TAG_compile_unit
- DW_TAG_namespace
- DW_TAG_subprogram
  - DW_TAG_formal_parameter
- DW_TAG_base_type
Fragmenting DWARF (example)

- Linker concatenates like-named sections together.
- Result is smaller DWARF with no dead references.
- Same approach works for other debug sections.
Fragmenting DWARF Limitations

• Works for .debug_info, .debug_line, .debug_aranges, .debug_ranges and .debug_loc.
  • Other sections don’t have direct references to variables/functions.
  • Doesn’t work for DWARF v5 .debug_rnglists/.debug_loclists, due to usage of entry indexes.

• Doesn’t get rid of all “useless” information.
  • E.g. empty namespace tags, unused .debug_abbrev entries.

• Intermediate objects not valid DWARF…
  • … but consumers could be taught how to read them.
Performance (Output Size)

Results for a large game link, using a script to fragment inputs.

- Unfragmented, DWARF sections
- Fragmented, DWARF sections
- Unfragmented, other data
- Fragmented, other data

Output Size (MB) vs. GC Aggressiveness:
- No GC
- 1 (normal GC)
- 2
- 3
- 4
- 5
- 6
Performance (Output Size)

Results for a large game link, using a script to fragment inputs.

Artificially treat more sections as “dead” by ignoring relocations in liveness analysis.
Performance (Link Time)

- **Fragmented, no gc-sections**
- **Unfragmented, no gc-sections**
Performance (Link Time)

- Fragmented, no gc-sections
- Fragmented, gc-sections
- Unfragmented, no gc-sections
- Unfragmented, gc-sections

Graph showing the relationship between GC Aggressiveness and Link Time for different memory fragmentation and collection section configurations.
Caveats

• Results use a modified LLD to support:
  • Using SHF_LINK_ORDER without ordering anything.
    • Reordering would corrupt the debug data.
  • References sections in groups, from outside the group.
    • Illegal according to the ABI.
  • Could use Group Sections instead.

• Figures include LLD patch improving performance.
  • Avoids doing some unnecessary string comparisons for debug sections.
  • May not be 100% correct for all objects - needs further investigation.
Alternative Solutions

• Rewrite DWARF at link time.
  • What the Sony proprietary linker does for PlayStation® 4 .debug_line.
  • Theoretically what LTO could effectively do.
  • Investigated within LLD by Alexey Lapshin (https://reviews.llvm.org/D74169).
  • Slow, and not particularly within traditional linker’s feature set.
    • 8 times slower in Alexey’s initial prototype when linking clang.

• Post-link optimization
  • Wasted I/O.
  • Relies on being able to identify dead debug data without relocations.
  • See llvm-dwarfutil proposal (http://lists.llvm.org/pipermail/llvm-dev/2020-August/144579.html).

• Change DWARF structure in new standard
  • Doesn’t solve issue for existing standards.
Conclusion

• Fragmenting the sections adds a lot of overhead.
  • Profiling LLD suggests it is largely due to the cost of creating more input sections internally.
  • String matching makes things slow.
  • Time savings from writing less outweighed by this overhead.

• Big size savings available, if willing to pay link time cost.
  • The more dead code, the better the trade-off.
  • Should improve debugger load times.

• Future work:
  • Investigate debugger load time improvements.
  • Use ELF Group sections instead of SHF_LINK_ORDER.
  • Investigate LLD performance improvements for many input sections.
  • Implement script in MC.
Appendix: Duration/Size Changes vs Unfragmented

• Figures for fragmented approach as a percentage of the unfragmented approach:

<table>
<thead>
<tr>
<th>Relocations used for GC liveness analysis</th>
<th>Link Time</th>
<th>Size (total)</th>
<th>Size (debug data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No GC</td>
<td>248%</td>
<td>91%</td>
<td>88%</td>
</tr>
<tr>
<td>100% (normal GC)</td>
<td>240%</td>
<td>84%</td>
<td>79%</td>
</tr>
<tr>
<td>80%</td>
<td>232%</td>
<td>55%</td>
<td>45%</td>
</tr>
<tr>
<td>60%</td>
<td>236%</td>
<td>39%</td>
<td>30%</td>
</tr>
<tr>
<td>40%</td>
<td>232%</td>
<td>31%</td>
<td>23%</td>
</tr>
<tr>
<td>20%</td>
<td>234%</td>
<td>28%</td>
<td>21%</td>
</tr>
<tr>
<td>0%</td>
<td>234%</td>
<td>28%</td>
<td>20%</td>
</tr>
</tbody>
</table>