RogueWave SOFTWARE

Innovate with Confidence

# SAVING TIME AND SPACE WITH SPLIT DWARF

Everyone's heard that the universe is expanding. Why? Perhaps it's to make room for larger and larger application programs, and the disk space needed to store their debugging information. This paper describes a relatively new approach for the DWARF Debugging Standard.

# OVERVIEW

Brian Kernighan said, "Everyone knows that debugging is twice as hard as writing a program in the first place. So, if you're as clever as you can be when you write it, how will you ever debug it?[1] " Of course, the obvious answer is to use a debugger, but to do that the debugger needs complete debug information.

Compilers generate debug information to describe application programs written in compiled languages, such as C, C++, and Fortran. The information includes the names and addresses of objects such as functions and variables, source file names and line numbers, and type information for built-in types, structure definitions, arrays, pointers, and more.

Split DWARF is a new approach that dramatically shrinks the space required to store DWARF debug information, speeds-up application link times, reduces system memory and IO requirements, and speeds-up debugger startup. Split DWARF places the majority of the debug information for an "object file" (.o) in a separate "DWARF object file" (.dwo) that is not processed by the linker.

The Split DWARF innovation was driven by the fact that applications evolve toward larger sizes over time, which is especially true for applications under development for many years, or even decades. The growth can often be attributed to the addition of features in the application, but the evolution of programming languages and paradigms also contributes to growth. For example, the U.S. Department of Energy national laboratories are developing "performance portability layers," such as RAJA and Kokkos, that make heavy use of C++11 templates and lambda functions. Such frameworks boost programmer productive, but increase the amount of debugging information required to describe the program, which contains many template instantiations, inline functions, and complex class types spread over thousands of source modules.

Application growth also happens in "dusty deck" Fortran programs and other legacy applications, some decades old and still under development. For example, Rogue Wave Software has TotalView for HPC customers with applications structured as monolithic executables containing over 2GB of debug information. Other customers, using modular techniques, have applications with an aggregate of over 10GB of debug information spread across hundreds of shared libraries.

Prior to Split DWARF, an image file (executable or shared library) contained all the debug information within a single file. Typically, DWARF debug information is stored in the image file itself, along with the code and data segments for the program. Various techniques have been developed to reduce the size of DWARF debug information, such as, the "dwz" DWARF Compressor Tool, compressed debug information sections, and DWARF 4 Compression and Duplicate Elimination. Not to be confused with Split DWARF or DWARF compression techniques, there are "separate debug information file" techniques where the debug information for the entire image is stored (compressed or uncompressed) in a separate file, and the image file containing the code and data "links" to that separate file. The GNU Debug Link and Build ID approaches are examples of separate debug information schemes, but others exist, such as the Oracle Solaris 11 Ancillary Objects. These techniques have their place, but the fact of the matter is the linker must process all the debug information, which comes with a cost:

[1] "The Elements of Programming Style", Brian W. Kernighan, 2nd edition, chapter 2.

- **Slow link times.** When linking together many object files into a single image file, the linker reads the debug information in the object files, applies the relocation information associated with the debug information, and writes it back out to the image file. The costs include IO consumption (disk and network traffic), CPU time to apply the relocations, and disk space to store the image file, which includes all the debug information.
- **Out of memory.** Depending on the amount of physical memory, the linker might consume so much memory that it starts paging. Excessive paging can lead to thrashing, further slowing down the linker. Ultimately, it's possible to run out of virtual memory, which will cause the linker to fail. One TotalView customer's application is so large they don't have enough memory (physical or virtual) on their development systems to link their application with full debug information. They must compile a subset of source modules with debug information, which is a burden on the developer since the set of modules that need to be debugged are frequently changing. The cost here is developer productivity.
- **Disk space consumption.** Traditionally, the compiler writes the debug information to the object file, and the linker copies it into the image file. That means there are two copies of the debug information, differing only in the relocations applied by the linker. The cost here is disk space consumption. In many cases, especially for C++, the debug information is about twice as large as the program's code and data. Simply eliminating one copy of the debug information can reduce disk consumption by 40 percent or more.

The use of Split DWARF is most appropriate during the implementation phase of the software development lifecycle, where the developer is routinely editing, compiling, linking, and debugging the application, and quick turnaround time affects the developer's productivity.

But Split DWARF may be inappropriate for production builds where the debug information is redistributed with the application and the full debug information for the program is spread across as many DWARF object files as there are source files. A way to solve this is by using a Split DWARF packager utility named "dwp" that combines the ".dwo" files into a single ".dwp" file for distribution.

# GNU DEBUGFISSION AND DWARF 5 SPLIT DWARF

Circa 2011, GNU started the GNU DebugFission Project. The goal of the DebugFission Project was to save time and disk space by eliminating most of the debug information that reaches the linker, using a "Split DWARF" approach. DebugFission was implemented as a DWARF "vendor extension" to DWARF 4, and was adopted, largely as-is, into the recently released DWARF 5 Standard.

The Split DWARF strategy is simple. During compilation, the compiler "splits the DWARF" into two parts: One part remains in the object (.o) file and the other is written to a corresponding DWARF object (.dwo) file. Lightweight "skeleton" DWARF debug information is included in the .o file, and full DWARF debug information is in the .dwo file. Since the linker processes only the .o files, not the .dwo files, the size of the object files processed is greatly reduced.

The notion of split debugging information is not new. In fact, the notion of Split DWARF is not a new idea. Over 20 years ago, Sun Microsystems implemented split debugging information for the STABS debugging format, where most the STABS remained in the object file, and lightweight "STABS index" information was written to the image file. When Apple switched from STABS to DWARF format, they implemented their own Split DWARF scheme for Mach-O object files, which they still use today. HP uses a different "+objdebug" Split DWARF variant on HP-UX. The Oracle Studio compilers produce another variant of Split DWARF, along with the original split STABS format.

The DebugFission variant of Split DWARF is available on Linux platforms using the GNU, Intel, and Clang compilers. Unfortunately, there is no implementation of the DWARF 5 variant of Split DWARF, though Red Hat is reportedly working on GCC prototype under the "–gdwarf-5" compiler option.

The remainder of this paper focuses on the DebugFission variant of Split DWARF, briefly describes how it works and how to build applications using Split DWARF.

## Debug Information Indexing

One important aspect of Split DWARF, or any split debugging format, is for the compiler and/or linker to generate good indexing information that the debugger can use to speed-up symbol table reading. Without it, the debugger has to open every .dwo (or .o) object file, and skim the DWARF information to determine where functions, variables, and types are defined. Skimming debug information out of potentially thousands of object files is more expensive than skimming the information out of a single image file.

DebugFission uses a section named ".gdb_index" to index the Split DWARF information. Currently, other than GDB itself, only the Linux "gold" linker (ld.gold) produces a ".gdb_index" section; therefore, the gold linker must be used when compiling Split DWARF and generating a .gdb_index section. Strictly speaking, it's not necessary to generate a .gdb_index when compiling with Split DWARF, but doing so allows faster debugger start up on large applications.

## Building for DebugFission Split DWARF

Figure 1 shows the basic build model for DebugFission Split DWARF on Linux for the GNU, Intel, or Clang compilers.

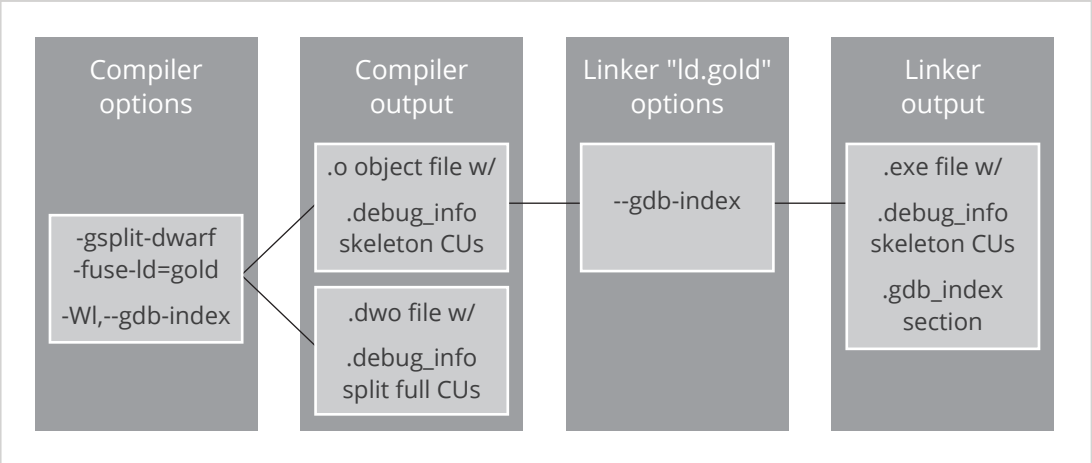


**Figure 1: The DebugFission Split DWARF model**

- **Compiling:**

  Use the "**-gsplit-dwarf**" compiler option to generate ".dwo" (DWARF object) files containing the full DWARF debug information, and a ".o" (object) file containing the code, data, and skeleton DWARF debug information.

  Note: As shown below, the DWARF debug information in the ".o" file points to the ".dwo" file, therefore the ".dwo" file must not be deleted to debug that module.

- **Linking:**

  Use the "**-fuse-ld=gold**" compiler option to use the gold linker (**ld.gold**).

  Use the "**-Wl,--gdb-index**" compiler option to pass the "**--gdb-index**" option to the gold linker.

  Note: The resulting executable or shared library image file will contain a "**.gdb_index**" section that the debugger can use for faster startup.

## A simple Split DWARF example

The easiest way to see how Split DWARF works is through an example. Figure 2 shows C++ code from www.cplusplus.com that uses code, which when compiled generates a relatively large number of symbols due to template expansions. The `makefile` shown in Figure 3 builds the sample program both with and without Split DWARF.

```
% cat map_empty.cxx
// http://www.cplusplus.com/reference/map/map/empty/
// map::empty
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;

  mymap['a']=10;
  mymap['b']=20;
  mymap['c']=30;

  while (!mymap.empty())
  {
    std::cout << mymap.begin()->first << " => " << mymap.begin()->second << '\n';
    mymap.erase(mymap.begin());
  }

  return 0;
}
%
```

**Figure 2: Sample C++ code using code**

```
% cat Makefile
all: map_empty.normal.exe map_empty.split-dwarf.exe
clean:
        rm *.o *.exe *.dwo

map_empty.normal.o: map_empty.cxx
        g++ -g -c -o $@ $<
map_empty.normal.exe: map_empty.normal.o
        g++ -o $@ $<

map_empty.split-dwarf.o: map_empty.cxx
        g++ -gsplit-dwarf -c -o $@ $<
map_empty.split-dwarf.exe: map_empty.split-dwarf.o
        g++ -o $@ $< -fuse-ld=gold -Wl,--gdb-index
%
```

**Figure 3: Code to build the sample program with and without Split DWARF**

The results of the build on a Fedora 25 x86_64 system are shown in Figure 4. Notice the following:

- The executable for the normal build is about 60 percent larger than the executable for the Split DWARF build.
- The object file for the normal build is about 75 percent larger than the object file for the Split DWARF build.
- The combined size of the Split DWARF built object and DWARF object files is smaller than the size of the normal build object file.

Even with this simple example, the Split DWARF build files are smaller than the normal build files. For large programs, the savings are even more dramatic.

```
% make
g++ -g -c -o map_empty.normal.o map_empty.cxx
g++ -o map_empty.normal.exe map_empty.normal.o
g++ -gsplit-dwarf -c -o map_empty.split-dwarf.o map_empty.cxx
g++ -o map_empty.split-dwarf.exe map_empty.split-dwarf.o -fuse-ld=gold -Wl,--gdb-index
% ls -l
total 572
-rw-r--r-- 1 jdelsign tss    355 Apr  7 15:20 Makefile
-rw-r--r-- 1 jdelsign tss    360 Apr  7 15:13 map_empty.cxx
-rwxr-xr-x 1 jdelsign tss 109656 Apr 10 13:30 map_empty.normal.exe
-rw-r--r-- 1 jdelsign tss 192512 Apr 10 13:30 map_empty.normal.o
-rw-r--r-- 1 jdelsign tss  70744 Apr 10 13:30 map_empty.split-dwarf.dwo
-rwxr-xr-x 1 jdelsign tss  67609 Apr 10 13:30 map_empty.split-dwarf.exe
-rw-r--r-- 1 jdelsign tss 110560 Apr 10 13:30 map_empty.split-dwarf.o
%
```

**Figure 4: Building the sample code on Linux (Fedora 25, x86_64)**

Figure 5 shows the debug information sections for the Split DWARF executable and DWARF object file. Notice the following:

- The **".debug_info"** section, which contain the DWARF debug information entries (DIEs) in the Split DWARF executable is small, occupying only 0x34 bytes.
- The **".debug_info.dwo"** section in the DWARF object file contains the bulk of the DIEs, occupying 0x6bbc bytes. The debug sections in a DWARF object file are all suffixed with .dwo.

```
% readelf -SW map_empty.split-dwarf.exe | egrep 'debug_|gdb_index|Nr'
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [29] .debug_addr       PROGBITS        0000000000000000 003d81 000378 00      0   0  1
  [30] .debug_info       PROGBITS        0000000000000000 0040f9 000034 00      0   0  1
  [31] .debug_abbrev     PROGBITS        0000000000000000 00412d 00001d 00      0   0  1
  [32] .debug_ranges     PROGBITS        0000000000000000 00414a 000690 00      0   0  1
  [33] .debug_line       PROGBITS        0000000000000000 0047da 000ea6 00      0   0  1
  [34] .debug_str        PROGBITS        0000000000000000 005680 000035 01  MS  0   0  1
  [35] .gdb_index        PROGBITS        0000000000000000 009c90 006b89 00      0   0  4
% readelf -SW map_empty.split-dwarf.dwo | egrep 'debug_|gdb_index|Nr'
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 1] .debug_info.dwo   PROGBITS        0000000000000000 000040 006bbc 00   E  0   0  1
  [ 2] .debug_abbrev.dwo PROGBITS        0000000000000000 006bfc 000ac8 00   E  0   0  1
  [ 3] .debug_line.dwo   PROGBITS        0000000000000000 0076c4 0003d2 00   E  0   0  1
  [ 4] .debug_str_offsets.dwo PROGBITS   0000000000000000 007a96 001074 00   E  0   0  1
  [ 5] .debug_str.dwo    PROGBITS        0000000000000000 008b0a 008604 00   E  0   0  1
%
```

**Figure 5: Sections in the Split DWARF executable image file**

Figure 6 shows a complete symbol table dump of the Split DWARF executable. It has exactly one DIE that is the "skeleton" compilation unit(CU) DIE that points to the DWARF object file. The lines labeled with the `DT_AT_GNU_*` attribute tags are GNU DWARF 4 extensions. The relevant attributes include:

- **DW_AT_GNU_dwo_name:** The name of the DWARF object file, which may include a directory path component.
- **DW_AT_comp_dir:** The standard DWARF attribute for the working directory path of the compiler when the module was compiled. The debugger uses this attribute to help locate the .dwo file.
- **DW_AT_GNU_dwo_id:** A unique 64-bit DWARF object ID value that must match the value in the corresponding ".dwo" file, as shown on the next page.

```
% readelf -wi map_empty.split-dwarf.exe
Contents of the .debug_info section:

  Compilation Unit @ offset 0x0:
   Length:        0x30 (32-bit)
   Version:       4
   Abbrev Offset: 0x0
   Pointer Size:  8
 <0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
    <c>   DW_AT_ranges      : 0x0
    <10>   DW_AT_low_pc      : 0x0
    <18>   DW_AT_stmt_list   : 0x0
    <1c>   DW_AT_GNU_dwo_name: (indirect string, offset: 0x0): map_empty.split-dwarf.
dwo
    <20>   DW_AT_comp_dir    : (indirect string, offset: 0x1a): /home/jdelsign/
split-dwarf
    <24>   DW_AT_GNU_pubnames: 1
    <24>   DW_AT_GNU_addr_base: 0x0
    <28>   DW_AT_GNU_dwo_id  : 0xa35fb8a809fe160e
    <30>   DW_AT_GNU_ranges_base: 0x0

%
```

**Figure 6: Contents of the ".debug_info" section in the Split DWARF executable file**

Figure 7 shows the symbol table dumps for the .dwo file. The first command shows there are 14,028 lines of DWARF information for all the symbols contained within the CU, significantly larger than the Split DWARF executable shown in Figure 6. The second command shows the "full split DWARF" CU DIE that provides the attributes for the source file, including the compiler that produced the DWARF, language, source file name, compilation working directory, and 64-bit DWARF object ID.

```
% readelf -wi map_empty.split-dwarf.dwo | wc -l
14028
% readelf -wi map_empty.split-dwarf.dwo | head -13
Contents of the .debug_info.dwo section:

  Compilation Unit @ offset 0x0:
   Length:        0x6bb8 (32-bit)
   Version:       4
   Abbrev Offset: 0x0
   Pointer Size:  8
 <0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
    <c>   DW_AT_producer    : (indexed string: 0x2b4): GNU C++14 6.3.1 20161221 (Red
Hat 6.3.1-1) -mtune=generic -march=x86-64 -gsplit-dwarf
    <e>   DW_AT_language    : 4 (C++)
    <f>   DW_AT_name        : (indexed string: 0x2a2): map_empty.cxx
    <11>   DW_AT_comp_dir    : (indexed string: 0x236): /home/jdelsign/split-dwarf
    <13>   DW_AT_GNU_dwo_id  : 0xa35fb8a809fe160e
%
```

**Figure 7: Contents of the ".debug_info.dwo" section in the DWARF object file**

Figure 8 shows a dump of the **".gdb_index"** section for the Split DWARF executable. As previously mentioned, the **".gdb_index"** section is not necessary in order to use Split DWARF (because most debuggers can handle Split DWARF without an index), but it is recommended for reducing debugger startup time. Briefly, the index section contains the following tables:

- A CU table containing entries that gives the compilations unit's DIE offset range. There's typically one entry per compilations unit, regardless of whether the CU is a skeleton CU or full DWARF CU. The entries in the CU table are referenced by other **".gdb_index"** tables using the CU table index.
- A TU table that holds "Type CUs", like the "CU table" above.
- An address table for by-address lookups, currently only for functions. Entry contains an address range for a function defined in the CU, and a CU table index. The debugger can quickly look up an address in this table to determine CU where the function is defined.
- A symbol table for by-name lookups of functions, variables, types, and other objects. The table is organized as a set of hash table entries, hashed by the canonical (demangled) name of the object. Each entry contains the name of the object, a CU table index, and an indication of whether the object is global or static, and a type, variable, or function. The debugger can quickly look up a name in this table to determine CU where the object is defined. Note that sometimes the linker doesn't generate a symbol table, which renders the entire index useless to the debugger, requiring the debugger to skim the contents of all the DWARF object files. Fortunately, this seems to happen only with small applications.

```
% readelf --debug-dump=gdb_index map_empty.split-dwarf.exe
Contents of the .gdb_index section:
Version 7

CU table:
[  0] 0x0 - 0x33

TU table:

Address table:
0000000000400ca6 0000000000400e4b 0
0000000000400e4b 0000000000400e5d 0
0000000000400e5d 0000000000400e6c 0
…snip…
0000000000402676 0000000000402690 0
0000000000402690 00000000004026aa 0
00000000004026aa 00000000004026e9 0

Symbol table:
[  2] std::move<char&>: 0 [global, function]
[  3] __gnu_cxx::new_allocator<std::_Rb_tree_node<std::pair<char const, int> > >::~new_
allocator: 0 [global, function]
[  4] std::_V2: 0 [global, type]
…snip…
[1015] std::allocator_arg: 0 [static, variable]
[1019] __gnu_cxx::__numeric_traits_integer<int>: 0 [global, type]
[1020] __gnu_cxx::__ops: 0 [global, type]
%
```

**Figure 8: Contents of the ".gdb_index" section in the Split DWARF executable file**

Finally, Figure 9 shows a debugging session on the sample Split DWARF executable file. This debugging session can be run on TotalView and CodeDynamics.

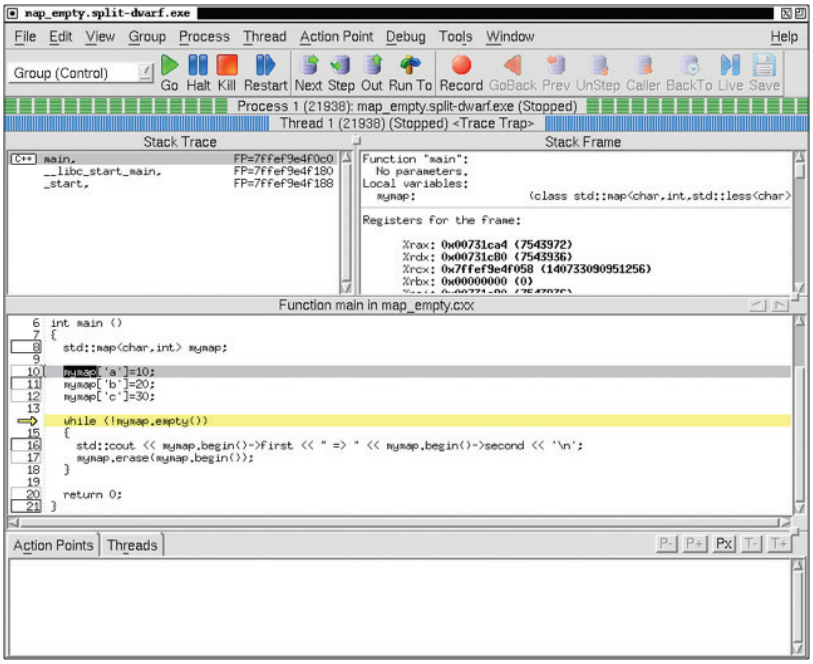


**Figure 9: A debugging session on the sample Split DWARF executable file.**

# TOTALVIEW AND CODEDYNAMICS

Rogue Wave dynamic analysis products, including TotalView and CodeDynamics, are mature, robust, scalable, and highly feature-rich debuggers supporting the C, C++, and Fortran programming languages, and hybrid mixtures of parallel programming paradigms, such as MPI, OpenMP, pthreads, CUDA, OpenACC, UPC, CAF, Global Arrays, SHMEM, and more.

The dynamic analysis products provide a graphical user interface (GUI) and command-line interface (CLI) for interactive debugging that allows users to debug large numbers of processes and threads consisting of multiple languages and parallel-programming paradigms, all from a single point of control. Standard debugger features include: Starting and stopping threads, processes, and groups of threads or processes; source and instruction-level step and step-over; breakpoints and conditional breakpoints; display of source code, variables, arrays, memory blocks, and registers; and modifying program state. The CLI is a programmable Tcl interpreter that supports the creation of user-defined utilities to extend debugger functionality, and provides a scripting language that allows for automated operation. TVScript and MemScript are frameworks for non-interactive batch debugging. Using an “event action” model, a user defines a series of events that may occur within the program and the actions to take when an event occurs. Data is logged to a set of output files for review when the batch job has completed.

TotalView and CodeDynamics offer many other features, including: A C, C++, and Fortran expression evaluator; data watchpoints; data visualizers; the ability to view the elements of STL collection classes (map, set, array, list, string, etc.); display aggregation; memory debugging; core file debugging; asynchronous thread control; reverse debugging; heterogeneous debugging; and extensive help and documentation.

For more information on how TotalView for HPC and CodeDynamics can help identify and resolve your runtime issues faster, visit roguewave.com.


Rogue Wave provides software development tools for mission-critical applications. Our trusted solutions address the growing complexity of building great software and accelerates the value gained from code across the enterprise. The Rogue Wave portfolio of complementary, cross-platform tools helps developers quickly build applications for strategic software initiatives. With Rogue Wave, customers improve software quality and ensure code integrity, while shortening development cycle times. roguewave.com