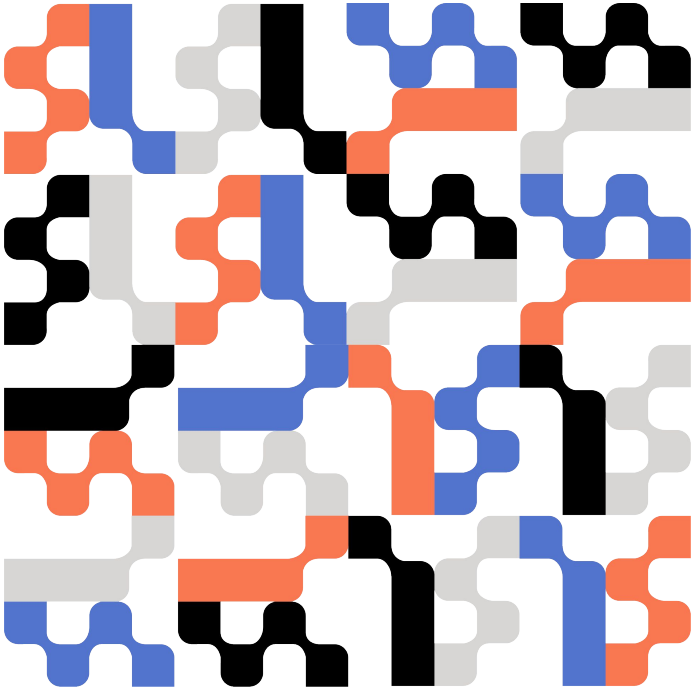
MODUS CREATE | TWEAG by Modus Create

Debug fission

Separating debug symbols from executables



Johan Herland – @jherland

FOSDEM 2025

What is this talk about?

- **Debug fission**
  - aka. “Split DWARF”
- with **ELF** files
- on **Linux**
- using **GCC**
- and the **gold** linker

- LLVM/clang works too, with minor adjustments
- Other linkers are available (mold, lld)
- YMMV, etc.

- Related, but not covered here:

  Compressed debug symbols:
  - Compile-time: GCC’s `-gz`
  - Link time: `--compress-debug-sections`
  - The `dwz` tool.

Debug symbols

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

```
$ g++ hello.cpp -o hello.default
$ gdb ./hello.default
GNU gdb (GDB) 13.1
[...]
Reading symbols from ./hello.default...
(No debugging symbols found in ./hello.default)
(gdb) br main
Breakpoint 1 at 0x4010a0
(gdb) run
Starting program: [...]/hello.default
[...]
Breakpoint 1, 0x00000000004010a0 in main ()
(gdb) list
No symbol table is loaded.  Use the "file" command.
```

```
$ g++ -g hello.cpp -o hello.with-g
$ gdb ./hello.with-g
[...]
Reading symbols from ./hello.with-g...
(gdb) br main
Breakpoint 1 at 0x4010a0: file hello.cpp, line 4.
(gdb) run
Starting program: [...]/hello.with-g
[...]
Breakpoint 1, main () at hello.cpp:4
4           std::cout << "Hello, world!" << std::endl;
(gdb) list
1       #include <iostream>
2
3       int main() {
4           std::cout << "Hello, world!" << std::endl;
5           return 0;
6       }
```

Debug symbols

```
$ ls -l hello.*
-rw-r--r-- 1 jherland users    97 Jan  1 00:00 hello.cpp
-rwxr-xr-x 1 jherland users  8280 Jan  1 00:00 hello.default
-rw-r--r-- 1 jherland users 29744 Jan  1 00:00 hello.o
-rwxr-xr-x 1 jherland users 31560 Jan  1 00:00 hello.with-g
```

+280%

```
$ readelf --sections --wide ...
```

```
+  [28] .debug_info       PROGBITS        0000000000000000 001023 002c66 00      0   0  1
+  [29] .debug_abbrev     PROGBITS        0000000000000000 003c89 0007b9 00      0   0  1
+  [30] .debug_loclists   PROGBITS        0000000000000000 004442 00010a 00      0   0  1
+  [31] .debug_aranges    PROGBITS        0000000000000000 00454c 000050 00      0   0  1
+  [32] .debug_rnglists   PROGBITS        0000000000000000 00459c 00007f 00      0   0  1
+  [33] .debug_line       PROGBITS        0000000000000000 00461b 000242 00      0   0  1
+  [34] .debug_str        PROGBITS        0000000000000000 00485d 001b62 01  MS  0   0  1
+  [35] .debug_line_str   PROGBITS        0000000000000000 0063bf 0004e1 01  MS  0   0  1
```

What's the problem?

- Space used ⇒ Time used
- At build-time
  - Generating debug symbols
  - Copied into intermediate build artifacts (object files, static libs, executables)
- At install/run time
  - Bigger release artifacts ⇒ Longer transfer times
  - Bigger executables ⇒ More memory used
- Significant overhead remains when we scale up to real-world projects.
  - Intermediate build artifacts grow by an order of magnitude when enabling debug symbols.

- How often do you actually need debug symbols vs. how much do they cost to build?

Stripped executables

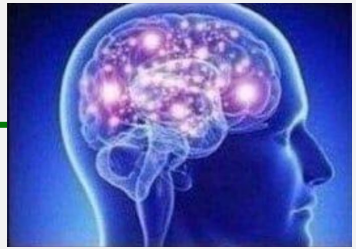
```
$ strip hello.with-g -o hello.stripped
$ ls -l hello.*
-rw-r--r-- 1 jherland users    97 Jan  1 00:00 hello.cpp
-rwxr-xr-x 1 jherland users  8280 Jan  1 00:00 hello.default
-rw-r--r-- 1 jherland users 29744 Jan  1 00:00 hello.o
-rwxr-xr-x 1 jherland users  6368 Jan  1 00:00 hello.stripped
-rwxr-xr-x 1 jherland users 31560 Jan  1 00:00 hello.with-g
```

-23%

```
$ gdb ./hello.stripped
[...]
Reading symbols from ./hello.stripped...
(No debugging symbols found in ./hello.stripped)
(gdb) br main
Function "main" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) br *0x4009e0
Breakpoint 1 at 0x4009e0
(gdb) run
Starting program:
/home/jherland/code/debug_fission_experiment/hello.stripped
[...]
Breakpoint 1, 0x00000000004009e0 in ?? ()
(gdb) list
No symbol table is loaded.  Use the "file" command.
```

Stripped + unstripped, best of both worlds?

```
$ gdb ./hello.stripped
[...]
Reading symbols from ./hello.stripped...
(No debugging symbols found in ./hello.stripped)
(gdb) symbol-file ./hello.with-g
Reading symbols from ./hello.with-g...
(gdb) br main
Breakpoint 1 at 0x4009e0: file hello.cpp, line 4.
(gdb) run
Starting program: /home/jherland/code/debug_fission_experiment/hello.stripped
[...]
Breakpoint 1, main () at hello.cpp:4
4           std::cout << "Hello, world!" << std::endl;
(gdb) list
1       #include <iostream>
2
3       int main() {
4           std::cout << "Hello, world!" << std::endl;
5           return 0;
6       }
```



Stripped + unstripped, best of both worlds?

```
$ objcopy --add-gnu-debuglink=hello.debug hello.stripped hello.stripped.debuglink
$ ls -l hello.stripped*
-rwxr-xr-x 1 jherland users 6368 Jan  1 00:00 hello.stripped
-rwxr-xr-x 1 jherland users 6464 Jan  1 00:00 hello.stripped.debuglink
```

```
$ gdb ./hello.stripped.debuglink
[...]
Reading symbols from ./hello.stripped.debuglink...
Reading symbols from /home/jherland/code/debug_fission_experiment/hello.debug...
(gdb) br main
Breakpoint 1 at 0x4009e0: file hello.cpp, line 4.
(gdb) run
Starting program:
/home/jherland/code/debug_fission_experiment/hello.stripped.debuglink
[...]
Breakpoint 1, main () at hello.cpp:4
4           std::cout << "Hello, world!" << std::endl;
(gdb) list
1       #include <iostream>
2
3       int main() {
4           std::cout << "Hello, world!" << std::endl;
5           return 0;
6       }
```

Stripped + unstripped, best of both worlds?

- Build with debug symbols, then strip
- Distribute stripped executable, keeps unstripped archived for future debugging
- Pros:
  - Smallest possible release package
  - Still debuggable (as long as we can retrieve the unstripped executable)
- Cons:
  - Must first generate unstripped artifacts
    - object files
    - intermediate archives
    - executable
  - Then strip the final executable.
  - (What if we never need to debug most executables?)

How to decrease build time?

Rather than stripping executables after the expensive debug build is already done,

1. Can we somehow split off debug symbols *while* we are compiling?

2. And still keep the separate debug symbols around for debugging later?

Debug fission

```
$ g++ -g -gsplit-dwarf -c hello.cpp -o hello.split.o
$ ls -l *o
-rw-r--r-- 1 jherland users 29744 Jan  1 00:00 hello.o
-rw-r--r-- 1 jherland users 20328 Jan  1 00:00 hello.split.dwo
-rw-r--r-- 1 jherland users 10640 Jan  1 00:00 hello.split.o
```

+4%

```
$ readelf --debug-dump hello.split.o
The .debug_info section contains link(s) to dwo file(s):

  Name:      hello.split.dwo
  Directory: /home/jherland/code/debug_fission_experiment


hello.split.o: Found separate debug object file: /home/[...]/hello.split.dwo

Contents of the .debug_addr section (loaded from hello.split.o):
[...]
Contents of the .debug_info section (loaded from hello.split.o):
[...]
[9 more sections loaded from hello.split.o...]
Contents of the .debug_info.dwo section (loaded from /home/[...]/hello.split.dwo):
[...]
Contents of the .debug_abbrev.dwo section (loaded from /home/[...]/hello.split.dwo):
[...]
[5 more sections loaded from hello.split.dwo...]
```

Debug fission, linking

```
$ g++ -g -gsplit-dwarf -c hello.cpp -o hello.split.o
$ ls -l *o
-rw-r--r-- 1 jherland users 29744 Jan  1 00:00 hello.o
-rw-r--r-- 1 jherland users 20328 Jan  1 00:00 hello.split.dwo
-rw-r--r-- 1 jherland users 10640 Jan  1 00:00 hello.split.o
```

```
$ g++ -fuse-ld=gold hello.split.o -o hello.split
$ ls -l hello.split hello.with-g
-rwxr-xr-x 1 jherland users 19984 Jan  1 00:00 hello.split
-rwxr-xr-x 1 jherland users 31560 Jan  1 00:00 hello.with-g
```

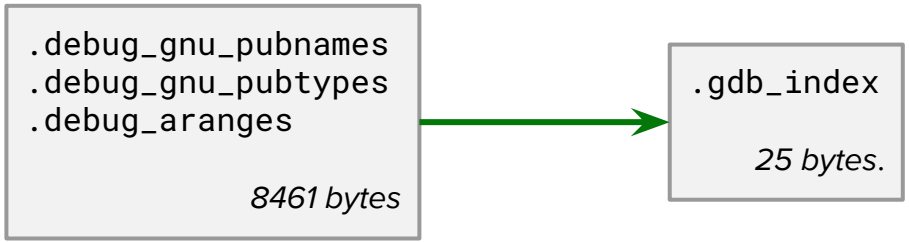
```
$ readelf --debug-dump hello.split
The .debug_info section contains link(s) to dwo file(s):

  Name:      hello.split.dwo
  Directory: /home/jherland/code/debug_fission_experiment


hello.split: Found separate debug object file: /home/[...]/hello.split.dwo
[...]
```

Another useful link option: `--gdb-index`

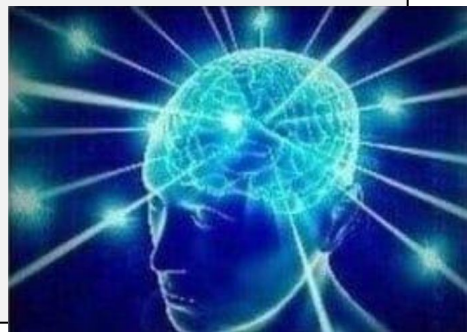
```
$ g++ -fuse-ld=gold -Wl,--gdb-index hello.split.o -o hello.split.gdbindex
$ ls -l hello.split hello.split.gdbindex hello.with-g hello.stripped
-rwxr-xr-x 1 jherland users 19984 Jan  1 00:00 hello.split
-rwxr-xr-x 1 jherland users 11377 Jan  1 00:00 hello.split.gdbindex
-rwxr-xr-x 1 jherland users  6368 Jan  1 00:00 hello.stripped
-rwxr-xr-x 1 jherland users 31560 Jan  1 00:00 hello.with-g
```



Consolidating .dwo files

```
$ dwp --exec hello.split.gdbindex
$ ls -l hello.split.dw*
-rw-r--r-- 1 jherland users 20328 Jan  1 00:00 hello.split.dwo
-rw-r--r-- 1 jherland users 57416 Jan  1 00:00 hello.split.gdbindex.dwp
```

```
$ rm *.dwo
$ gdb hello.split.gdbindex
[...]
Reading symbols from hello.split.gdbindex...
(gdb) br main
Breakpoint 1 at 0x400a00: file hello.cpp, line 4.
(gdb) run
Starting program: /home/jherland/code/debug_fission_experiment/hello.split.gdbindex
[...]
Breakpoint 1, main () at hello.cpp:4
4           std::cout << "Hello, world!" << std::endl;
(gdb) list
1       #include <iostream>
2
3       int main() {
4           std::cout << "Hello, world!" << std::endl;
5           return 0;
6       }
```



Summary of debug fission

Results:

+ A debuggable executable, only slightly larger than a stripped executable.
+ Accompanying `.dwp` package of debug symbols.
+ Distribute/deploy executable on its own, supply `.dwp` file when you need to debug.
+ Faster build times due to smaller linker inputs, and linking debug information separately.

Recap:

1. Compiler produces *two* output files: `.o` + `.dwo`.
2. `.o` file carries a reference to the corresponding `.dwo` file, forwarded by linker into the final executable.
3. The `.dwo` files can also be “linked” together into a `.dwp` package, containing *all* debug symbols for an executable.
4. GDB can find debug symbols in both `.dwp` and `.dwo` files, as long as either is available to GDB.
5. Using `-Wl,--gdb-index` allows further debugging optimizations to be precomputed into the final executable.

Integration into larger build systems

1. CMake
2. Bazel

CMake: First steps

```
cmake_minimum_required(VERSION 3.25)
project(debug_fission_experiment)
add_executable(hello hello.cpp)
```

```
add_link_options(-fuse-ld=gold)
set(CMAKE_BUILD_TYPE Debug)
```

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -gsplit-dwarf")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -gsplit-dwarf")
set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -Wl,--gdb-index")
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -Wl,--gdb-index")
```

CMake: Producing the .dwp debug package

```
cmake_minimum_required(VERSION 3.25)
project(debug_fission_experiment)
add_link_options(-fuse-ld=gold)
set(CMAKE_BUILD_TYPE Debug)
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -gsplit-dwarf")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -gsplit-dwarf")
set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -Wl,--gdb-index")
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -Wl,--gdb-index")

add_executable(hello hello.cpp)
```

```
find_program(DWP_TOOL dwp)
function(add_executable target_name)
    # Call the original function
    _add_executable(${target_name} ${ARGN})
    set(out_dwp "${target_name}.dwp")
    add_custom_command(TARGET ${target_name}
        POST_BUILD
        COMMAND ${DWP_TOOL} --exec ${target_name} -o ${out_dwp}
        WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
        COMMENT "Linking debug package ${out_dwp}"
        VERBATIM
        )
endfunction()
```

+ CMake issue #21179: Natively support split dwarf
+ “hacky workarounds”, “fragile”...

Bazel

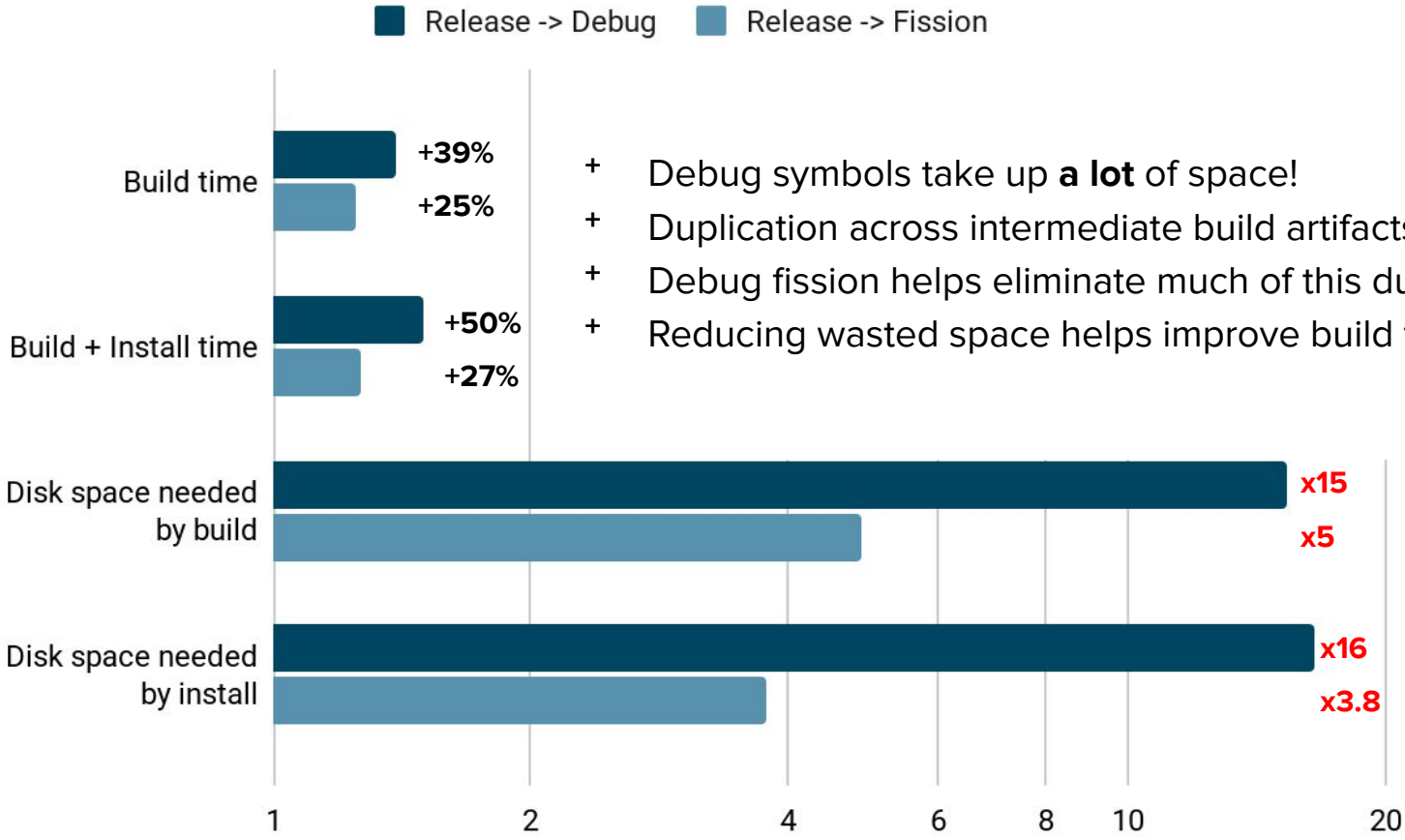
- Since version 6
- `--fission=yes`
- Conditional on the underlying toolchain configuration:
  - `per_object_debug_info` toolchain feature.

- `bazel build --fission=yes //path/to:executable`
- `bazel build --fission=yes //path/to:executable.dwp`

Some numbers from a larger project

- Building the LLVM compiler (yeah, I known this is the GCC track, sorry...)
- CMake build provides `LLVM_USE_SPLIT_DWARF` option to enable debug fission
- Compare three separate builds:
  a. “Release”: no debug symbols, our baseline
  b. “Debug”: with debug symbols, but no debug fission
  c. “Fission”: with debug symbols and debug fission enabled

Three LLVM builds compared



- \+ Debug symbols take up **a lot** of space!
- \+ Duplication across intermediate build artifacts.
- \+ Debug fission helps eliminate much of this duplication.
- \+ Reducing wasted space helps improve build times.

Conclusions

- Debug fission can save both time and space
- But adds complexity to you build process
  - Especially if not already supported by your build system
- Is debug fission worth it? Depends…
  - Remember unstripped + stripped:
    - *least* overhead in terms of release size,
    - but *most* overhead in terms of build time: full debug build, then strip.
  - If you struggle with build space/time overhead: try debug fission!

More resources

- Primary inspiration:
  - Improving C++ Builds with Split DWARF, by Martin Richtarsky
  - Building for Linux, the smart way, by Leszek Godlewski
- GCC Wiki:
  - https://gcc.gnu.org/wiki/DebugFission
  - https://gcc.gnu.org/wiki/DebugFissionDWP
- GDB docs on using `--only-keep-debug` and `--add-gnu-debuglink`:
  - https://www.sourceware.org/gdb/onlinedocs/gdb/Separate-Debug-Files.html
- Also:
  - Linux Debuginfo Formats: DWARF, ELF, dwo, dwp - What are They All?, by Greg Law
  - Shrinking a Shared Library, by Serge “sans Paille” Guelton
  - Tiny ELF Files: Revisited in 2021, by Nathan Otterness

THANK YOU!

Slides

- This talk ~~could have been~~ is also a blog post!

- Tweag blog: tweag.io/blog
- Tweag: tweag.io
- Modus Create: moduscreate.com

@jherland