Last week's blog entry discussed relocations and how they apply to the RISC-V toolchain. This week we'll be delving a bit deeper into the RISC-V linker to discuss linker relaxation, a concept so important it has greatly shaped the design of the RISC-V ISA. Linker relaxation is a mechanism for optimizing programs at link-time, as opposed to traditional program optimization which happens at compile-time. This blog will follow an example linker relaxation through the toolchain, demonstrate an example of how linker relaxations meaningfully improve the performance of a real program and introduce a new RISC-V relocation. We'll shy away from discussing the impact of linker relaxations on the RISC-V ISA, until another blog entry.

Just like last time, we'll start with a simple C test program that's not linked against anything else. This program won't perform a sane computation, the goal is just that it's simple enough to get the point across. I'll skip the assembly this time: as this post is about the linker, we can't really discuss anything until we get to an object file. Since you're now an expert in the toolchain, I'll just blast out some commands here:

```
$ cat test.c
int func(int a) __attribute__((noinline));
int func(int a) {
  return a + 1;
}

int _start(int a) {
  return func(a);
}
$ riscv64-unknown-linux-gnu-gcc test.c -o test -O3
$ riscv64-unknown-linux-gnu-objdump -d -r test.o
test.o:     file format elf64-littleriscv
Disassembly of section .text:

0000000000000000 <func>:
   0:   2505                    addiw   a0,a0,1
   2:   8082                    ret

0000000000000004 <_start>:
   4:   00000317                auipc   ra,0x0
                        4: R_RISCV_CALL func
                        4: R_RISCV_RELAX        *ABS*
   8:   00030067                jr      ra
```

You can now see a new RISC-V relocation: `R_RISCV_CALL`. This relocation sits between an `auipc` and a `jalr` instruction (here disassembled as the `jr` shorthand as this is a tail call) and points to the symbol that should be the target of the jump, in this case the `func` symbol. The `R_RISCV_CALL` relocation is paired with a `R_RISCV_RELAX` relocation, which allows the linker to relax this relocation pair -- the whole point of this blog!

In order to understand relaxation, we first must examine the RISC-V ISA a bit. In the RISC-V ISA there are two unconditional control transfer instructions: `jalr`, which jumps to an absolute address as specified by an immediate offset from a register; and `jal`, which jumps to a pc-relative offset as specified by an immediate. The only differences between the `auipc`+`jalr` pair in this object file and a single `jal` are that the pair can address a 32-bit signed offset from the current PC while the `jal` can only address a 21-bit signed offset from the current PC, and that the `jal` instruction is half the size (which is a good proxy for twice the speed).

As the compiler cannot know if the offset between `_start` and `func` will fit within a 21-bit offset, it is forced to generate the longer call. We don't want to impose this cost in cases where it's not necessary, so we instead optimize this case in the linker. Let's look at the executable to see the result of linker relaxation:

```
$ riscv64-unknown-linux-gnu-objdump -d -r test
test:     file format elf64-littleriscv

Disassembly of section .text:
```

```
0000000000010078 <func>:
   10078:       2505                    addiw   a0,a0,1
   1007a:       8082                    ret

000000000001007c <_start>:
   1007c:       ffdff06f                j       10078 <func>
```

As you can see, the linker knows that the call from `_start` to `func` fits within the 21-bit offset of the `jal` instruction and converts it to a single instruction.

The RISC-V Implementation of Linker Relaxation

While the concept of linker relaxation is fairly straight-forward, there are a lot of tricky details that need to be done correctly in order to ensure the linker produces the correct symbol addresses everywhere. To the best of my knowledge, the RISC-V BFD port make the most aggressive use of linker relaxations: essentially no `.text` section symbol addresses are known until link time. This has a few interesting side effects:

- `.align` directives must be handled by the linker for any relaxable sections.
- Debug information must be emitted twice: once by the compiler for the object file and once again by the linker for the executable.

All these points probably warrant blog posts of their own -- some of these are planned, others require me to fix some bugs before I feel comfortable talking about them :).

The actual implementation of linker relaxation is, as you'd expect, fairly esoteric. The code lives in `_bfd_riscv_relax_section` inside `binutils-gdb/bfd/elfnn-riscv.c`, which looks roughly like the following:

```
_bfd_riscv_relax_section:
  if section shouldn't be relaxed:
    return DONE
  for each relocation:
    if relocation is relaxable:
      store per-relocation function pointer
    read the symbol table
    obtain the symbol's address
    call the per-relocation function
```

Essentially, all it's doing is some shared bookkeeping code and then calling a relocation-specific function to actually relax the relocation. The relax functions all look somewhat similar, so I'll show an example of the function that relaxes `R_RISCV_CALL` relocation that was discussed above

```
_bfd_riscv_relax_call:
  compute a pessimistic address range
  if relocation doesn't fit into a UJ-type immediate:
    return DONE
  compute offsets for various short jumps
  if RVC is enabled and the relocation fits in a C.JAL:
    convert the jump to c.jal
```

```
  if relocation fits in an JAL:
    convert the jump to a jal
  if call target is near absolute address 0:
    convert the jump to a x0-offset JALR
  delete the JALR, as it's not necessary anymore
```

While this specific function only relaxes the `R_RISCV_CALL` relocation, it follows the pattern that most of the implementations of the relaxation functions do:

```
generic_relax_function:
  add some slack to the address, as all addresses can move
  for each possible instruction to shorten the relocation:
    if possible instruction can fit the target address:
      replace the relocation
      cleanup
      return DONE
  return DONE
```

There's one of these functions for each class of relocations that the RISC-V toolchain knows how to relax:

- `_bfd_riscv_relax_call`: relaxes two-instruction `auipc`+`jalr` sequences via the `R_RISCV_CALL` and `R_RISCV_CALL_PLT` relocations.
- `_bfd_riscv_relax_lui`: relaxes two-instruction `lui`+`addi`-like sequences via the `R_RISCV_HI20`+`R_RISCV_LO12_I`-like relocation pairs. The second instruction/relocation can be any of the various instructions that matches a `R_RISCV_LO12_I` or `R_RISCV_LO12_S` relocation (`addi`, `lw`, `sd`, etc).
- `_bfd_riscv_relax_pc`: Relaxes two-instruction `auipc`+`addi`-like sequences via the `R_RISCV_PCREL_HI20`'+'`R_RISCV_PCREL_LO12_I`-like relocation pairs. Much like the `lui` case there's a handful of relocation types possible for the second one, all of which are PC-relative.
- `_bfd_riscv_relax_tls_le`: Relaxes thread local storage references when using the local executable model. We'll talk about TLS is a later blog, as there's a lot going on here.
- `_bfd_riscv_relax_align`: Relaxes `.align` directives in text sections. This is another on we'll discuss later, but one specific interesting constraint here is that `R_RISCV_ALIGN` relocations must be relaxed for correctness, which means they're relaxed even when relaxations are otherwise disabled.

Relaxing Against the Global Pointer

It may seem like linker relaxation involves a huge amount of complexity for a small gain: we trade knowing no `.text` section symbol addresses until link time for shortening a few sequences by a single instruction. As it turns out, linker relaxation is very important for getting good performance on real code. For our first time looking at real code, we'll take a look at the Dhrystone benchmark -- in addition to being super simple, Dhrystone also spends a lot of time loading from global variables and therefore benefits very clearly from linker relaxation.

Let's take a look at the Dhrystone source code first. While it's a bit more complicated than the examples that have been present in this blog so far, if you look closely the code is actually

pretty straightforward. Here's the source for one Dhrystone function, along with the definitions of the various global variables it references:

```
/* Global Variables: */
Boolean         Bool_Glob;
char            Ch_1_Glob,
                Ch_2_Glob;

Proc_4 () /* without parameters */
/*******/
    /* executed once */
{
  Boolean Bool_Loc;

  Bool_Loc = Ch_1_Glob == 'A';
  Bool_Glob = Bool_Loc | Bool_Glob;
  Ch_2_Glob = 'B';
} /* Proc_4 */
```

As you can see, the code performs three accesses to global variables in order to do a simple comparison and a logical operation. While this might seem kind of silly, this is what a lot of the Dhrystone benchmark looks like. Since Dhrystone is pretty much the only benchmark that will actually run *everywhere* (SPECInt won't run on my wristwatch, for example), it's still used as the baseline for many microarchitectural comparisons so we need to make it go fast.

In order to understand the specific relaxation that's being performed in this case, it's probably best to start with the code the toolchain generates before this optimization, which I've copied below:

```
0000000040400826 <Proc_4>:
    40400826:   3fc00797                auipc   a5,0x3fc00
    4040082a:   f777c783                lbu     a5,-137(a5) # 8000079d <Ch_1_Glob>
    4040082e:   3fc00717                auipc   a4,0x3fc00
    40400832:   f7272703                lw      a4,-142(a4) # 800007a0 <Bool_Glob>
    40400836:   fbf78793                addi    a5,a5,-65
    4040083a:   0017b793                seqz    a5,a5
    4040083e:   8fd9                    or      a5,a5,a4
    40400840:   3fc00717                auipc   a4,0x3fc00
    40400844:   f6f72023                sw      a5,-160(a4) # 800007a0 <Bool_Glob>
    40400848:   3fc00797                auipc   a5,0x3fc00
    4040084c:   04200713                li      a4,66
    40400850:   f4e78a23                sb      a4,-172(a5) # 8000079c <Ch_2_Glob>
    40400854:   8082                    ret
```

As you can see, this function consists of 13 instructions, 4 of which are `auipc` instructions. All of these `auipc` instructions are used to calculate the addresses of global variables for a subsequent memory access, and all of these generated addresses are within a 12-bit offset of each other. If you're thinking "we only really need one of these `auipc` instructions", you're both right and wrong: while we could generate a single `auipc` (though that requires some GCC work we haven't done yet and is thus the subject of a future blog post), we can actually do one better and get by with *zero* `auipc` instructions!

If you've just gone and pored over the RISC-V ISA manual to find an instruction that loads `Ch_1_Glob` (which lives at `0x8000079D`) in a single instruction then you should give up now, as there isn't one. There is, of course, a trick -- it is common on register-rich, addressing-mode-poor ISAs to have a dedicated ABI register known as the global pointer that contains an address in the `.data` segment. The linker is then capable of relaxing accesses to global variables that live within a 12-bit signed offset from this value -- essentially we've just cached the `lui` in the global pointer register, optimizing this common code path.

In order to get a bit more visibility into how this works, let's take a look at a snippet of GCC's default linker script for RISC-V:

```
/* We want the small data sections together, so single-instruction offsets
   can access them all, and initialized data all before uninitialized, so
   we can shorten the on-disk segment size.  */
.sdata          :
{
  __global_pointer$ = . + 0x800;
  *(.srodata.cst16) *(.srodata.cst8) *(.srodata.cst4) *(.srodata.cst2) *(.srodata .srodata.*)
  *(.sdata .sdata.* .gnu.linkonce.s.*)
}
_edata = .; PROVIDE (edata = .);
. = .;
__bss_start = .;
.sbss           :
{
  *(.dynsbss)
  *(.sbss .sbss.* .gnu.linkonce.sb.*)
  *(.scommon)
```

As you can see, the magic `__global_pointer$` symbol is defined to point `0x800` bytes past the start of the `.sdata` section. The `0x800` magic number allows signed 12-bit offsets from `__global_pointer$` to address symbols at the start of the `.sdata` section. The linker assumes that if this symbol is defined, then the `gp` register contains that value, which it can then use to relax accesses to global symbols within that 12-bit range. The compiler treats the `gp` register as a constant so it doesn't need to be saved or restored, which means it is generally only written by `_start`, the ELF entry point. Here's an example from the RISC-V newlib port's `crt0.S` file

```
.option push
.option norelax
1:auipc gp, %pcrel_hi(__global_pointer$)
  addi  gp, gp, %pcrel_lo(1b)
.option pop
```

Note that we need to disable relaxations while setting `gp`, otherwise the linker would relax this two-instruction sequence to `mv gp, gp`

The actual implementation of the relaxation, which lives in `_bfd_riscv_relax_lui` and `_bfd_riscv_relax_pc`, is fairly boring. Like all the other relaxations, it performs some bounds checks, deletes the unused instruction and then converts the short-offset instruction to a different type. We may delve deeper into the implementation of various linker relaxations in future blog posts, but for now I'll just drop the relaxed output here to demonstrate it actually works:

```
00000000400003f0 <Proc_4>:
    400003f0:   8651c783                lbu     a5,-1947(gp) # 80001fbd <Ch_1_Glob>
    400003f4:   8681a703                lw      a4,-1944(gp) # 80001fc0 <Bool_Glob>
    400003f8:   fbf78793                addi    a5,a5,-65
    400003fc:   0017b793                seqz    a5,a5
    40000400:   00e7e7b3                or      a5,a5,a4
    40000404:   86f1a423                sw      a5,-1944(gp) # 80001fc0 <Bool_Glob>
    40000408:   04200713                li      a4,66
    4000040c:   86e18223                sb      a4,-1948(gp) # 80001fbc <Ch_2_Glob>
    40000410:   00008067                ret
```

12-bit Offsets aren't Enough for Anyone

Just to be clear: linker relaxations are an optimization for the common case. The linker transparently emits two-instruction addressing sequences for symbols that it cannot optimize. To demonstrate what happens when the linker can't relax a symbol, let's go through another example:

```
$ cat relax.c
long near;
long far[2];

long data(void) {
  return near | far;
}

int main() {
  return data();
}
$ riscv64-unknown-linux-gnu-gcc relax.c -O3 -o relax --save-temps
$ riscv64-unknown-linux-gnu-objdump -d relax.o
relax.o:     file format elf64-littleriscv

Disassembly of section .text:

0000000000000000 data:
   0:   000007b7                lui     a5,0x0
                        0: R_RISCV_HI20 near
                        0: R_RISCV_RELAX        *ABS*
   4:   0007b503                ld      a0,0(a5) # 0 data
                        4: R_RISCV_LO12_I       near
                        4: R_RISCV_RELAX        *ABS*
   8:   000007b7                lui     a5,0x0
                        8: R_RISCV_HI20 far
                        8: R_RISCV_RELAX        *ABS*
   c:   0007b783                ld      a5,0(a5) # 0 data
                        c: R_RISCV_LO12_I       far
                        c: R_RISCV_RELAX        *ABS*
  10:   8d5d                    or      a0,a0,a5
  12:   8082                    ret

Disassembly of section .text.startup:

0000000000000000 main:
   0:   1141                    addi    sp,sp,-16
```

```
   2:   e406                   sd      ra,8(sp)
   4:   00000097               auipc   ra,0x0
                       4: R_RISCV_CALL data
                       4: R_RISCV_RELAX        *ABS*
   8:   000080e7               jalr    ra
   c:   60a2                   ld      ra,8(sp)
   e:   2501                   sext.w  a0,a0
  10:   0141                   addi    sp,sp,16
  12:   8082                   ret
```

Here we can see three relocations groups: two `HI20`/`LO12` relocation pairs and a `CALL` relocation. In this case, the `CALL` relocation can be relaxed as can the `HI20`/`LO12` pair that references `near`, but the `HI20`/`LO12` pair that references `far` can't. In this case the linker still functions correctly, producing a single-instruction addressing sequence for the `near` symbol it can relax while just relocating the addressing sequence for the `far` symbol that it can't reference with a single instruction.

```
$ riscv64-unknown-linux-gnu-objdump -d -r relax
Disassembly of section .text:

0000000000010330 main:
   10330:       1141                    addi    sp,sp,-16
   10332:       e406                    sd      ra,8(sp)
   10334:       0b8000ef                jal     ra,103ec data
   10338:       60a2                    ld      ra,8(sp)
   1033a:       2501                    sext.w  a0,a0
   1033c:       0141                    addi    sp,sp,16
   1033e:       8082                    ret

00000000000103ec data:
   103ec:       8181b503                ld      a0,-2024(gp) # 12038 near
   103f0:       67e9                    lui     a5,0x1a
   103f2:       0407b783                ld      a5,64(a5) # 1a040 far
   103f6:       8d5d                    or      a0,a0,a5
   103f8:       8082                    ret
```

Though it might be a bit redundant at this point, I already had the following examples written out so I figured I'd just leave them here to be a bit more explicit:

```
--- relax.o
+++ relax
-      4:   00000097               auipc   ra,0x0
-                          4: R_RISCV_CALL         data
-                          4: R_RISCV_RELAX        *ABS*
-      8:   000080e7               jalr    ra
+  10334:   0b8000ef               jal     ra,103ec data
```

In the example above, we can see the `R_RISCV_CALL` relocation is requested. This relocation is defined to operate over an adjacent `auipc`/`jalr` pair, referencing a signed 32-bit PC-relative call target. In this case we were able to relax this instruction pair to a single `jal` instruction as the actual call target was within a 21-bit signed offset from the current PC. You'll find that almost all `R_RISCV_CALL` relocations will be relaxable, as most code expresses some amount of call locality.

```
--- relax.o
+++ relax
-      0:   000007b7                lui     a5,0x0
-                           0: R_RISCV_HI20         near
-                           0: R_RISCV_RELAX        *ABS*
-      4:   0007b503                ld      a0,0(a5) # 0 data
-                           4: R_RISCV_LO12_I       near
-                           4: R_RISCV_RELAX        *ABS*
+  103ec:   8181b503                ld      a0,-2024(gp) # 12038 near
```

In the example above, we can see a `R_RISCV_HI20`/`R_RISCV_LO12_I` relocation pair is requested. These relocations are each defined to operate over a single instruction: the `R_RISCV_HI20` relocates the 20-bit offset of a `lui` while the `R_RISCV_LO12_I` relocates the 12-bit offset of various I-type instructions (`ld` in this example). In this case we were able to relax this instruction pair to a single `ld` instruction, as the final symbol address was within a 12-bit offset of `gp`, the global pointer.

```
--- relax.o
+++ relax
-      8:   000007b7                lui     a5,0x0
-                           8: R_RISCV_HI20         far
-                           8: R_RISCV_RELAX        *ABS*
-      c:   0007b783                ld      a5,0(a5) # 0 data
-                           c: R_RISCV_LO12_I       far
-                           c: R_RISCV_RELAX        *ABS*
+  103f0:   67e9                    lui     a5,0x1a
+  103f2:   0407b783                ld      a5,64(a5) # 1a040 far
```

In the example above, we see another `R_RISCV_HI20`/`R_RISCV_LO12_I`, but this time we can't relax it as it's not within a 12-bit offset of `gp`. Note that we still generate the correct code for this case by filling out the relocations. You will get a link-time error whenever it is impossible to correctly relocate a requested relocation, as otherwise the linked executable wouldn't produce the correct answer.

Stay tuned, as there's a whole lot more to come on linker relaxations in future blog posts.