# A closer look at ARM code size

**Tilmann Scheller**
**Principal Compiler Engineer**
**t.scheller@samsung.com**

**Samsung Open Source Group**
**Samsung Research UK**

**EuroLLVM 2016**
**Barcelona, Spain, March 17 – 18, 2016**

# Overview

- Introduction
- ARM architecture
- Case study
- Summary

# Introduction

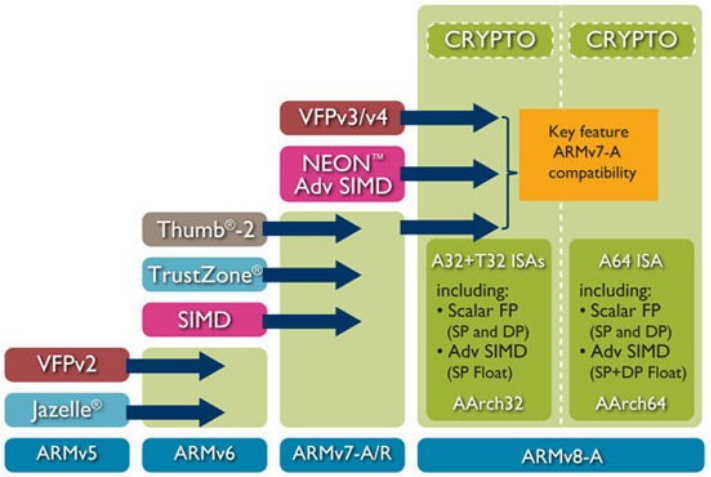
# Introduction

- Code size matters in the embedded space
- Find out how we are doing on ARM
- Comparison against GCC
- Focus on a specific application and compare the generated assembly code
- Try to find out what we need to change in LLVM to get better code size

# ARM architecture

# ARM architecture

- 32-bit/64-bit RISC architecture
- Load-store architecture
- Barrel shifter: **add r4, r3, r6, lsl #4**
- Powerful indexed addressing modes: **ldr r0, [r1, #4]!**
- Predication: **ldreq r3, [r4]**
- Family of 32-bit instruction sets evolved over time: ARM, Thumb, Thumb-2
- Focus on the Thumb-2 instruction set in this talk
- Instruction set extensions:
  - VFP
  - Advanced SIMD (NEON)



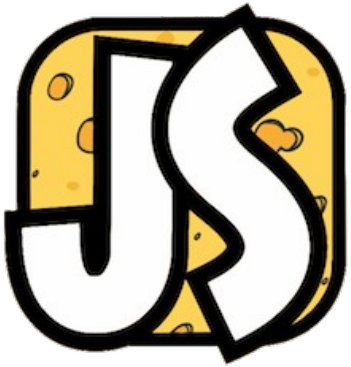
# Thumb-2 ISA

- Goal: Code density similar to Thumb, performance like original ARM instruction set
- Variable-length instructions (16-bit/32-bit)
- 16 32-bit GPRs (including PC and SP)
- 16 or 32 64-bit floating-point registers for VFP/NEON
- Conditional execution with IT (if-then) instruction

```
; if (r0 == r1)
cmp r0, r1
ite eq          ; ARM: no code ... Thumb: IT instruction
; then r0 = r2;
moveq r0, r2   ; ARM: conditional; Thumb: condition via ITE 'T' (then)
; else r0 = r3;
movne r0, r3   ; ARM: conditional; Thumb: condition via ITE 'E' (else)
; recall that the Thumb MOV instruction has no bits to encode "EQ" or "NE"
```

# Case study

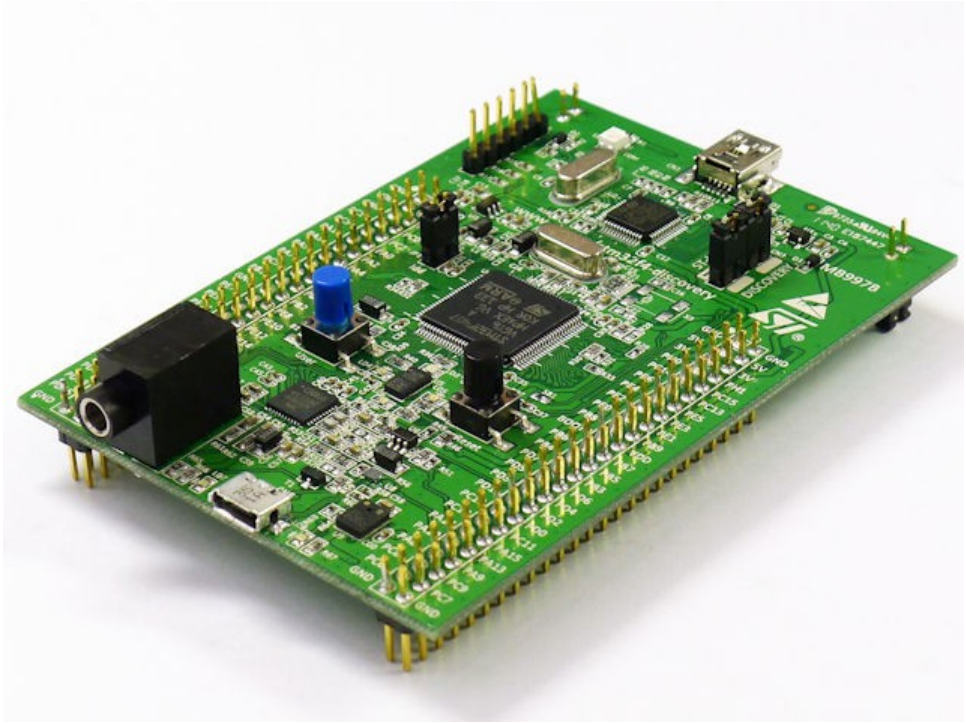
# JerryScript



- A really lightweight JavaScript engine
- Has a base footprint of 10KB of RAM
- Optimized for microcontrollers
- Written in C99
- Supports all of ECMAScript 5.1
- Open source (Apache 2.0 license)
- Developed by Samsung
- Goal: Want to compile with Clang but code size significantly higher

# Target hardware

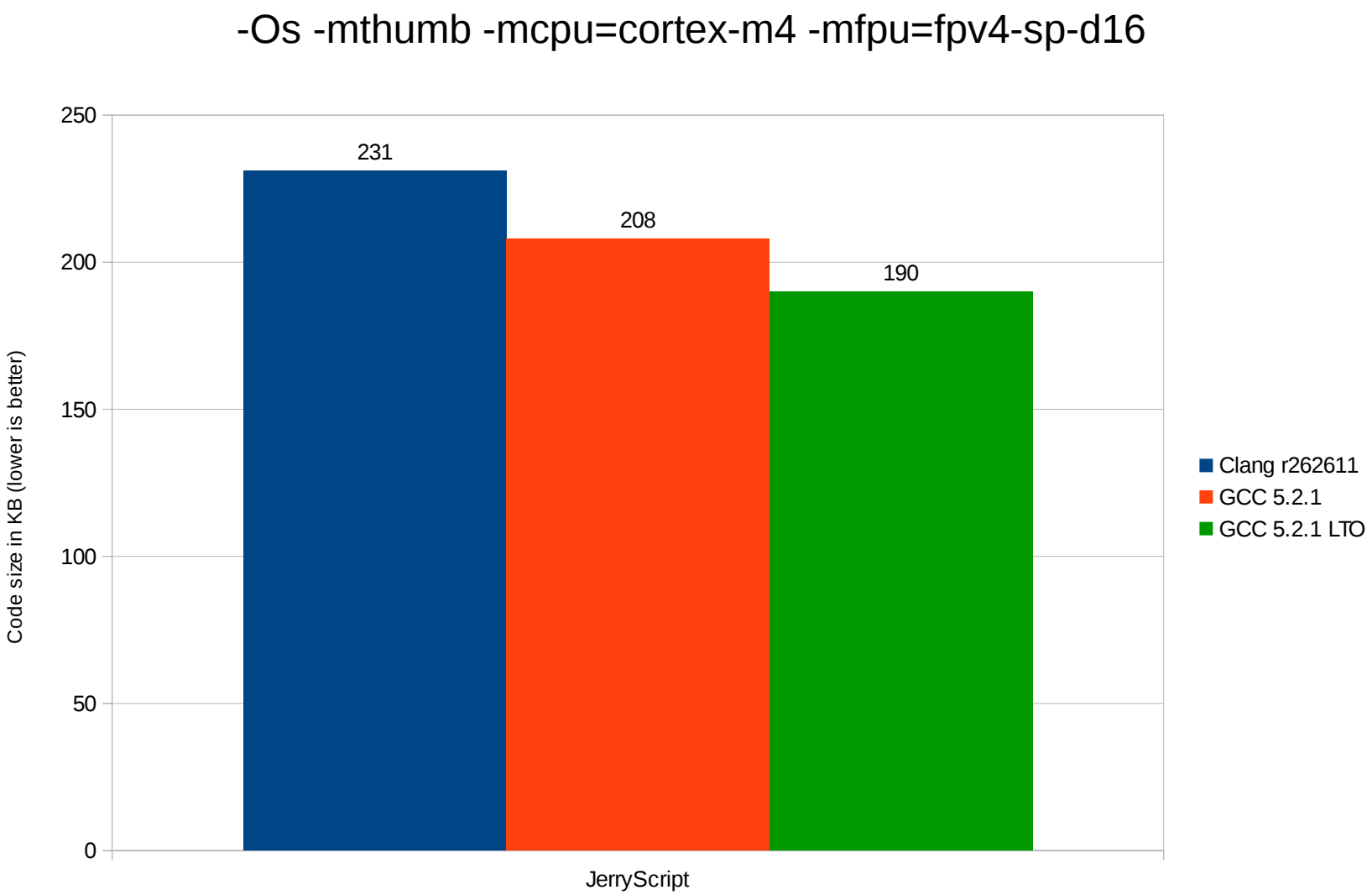
- STM32F4 developer board
- Cortex-M4F clocked at 168 MHz
- 192KB of RAM
- 1MB of flash memory



# Code size

- Compiling with a Clang trunk snapshot (r262611, March 3, 2016)
- Using GCC 5.2.1 (Linaro bare-metal toolchain)
- JerryScript trunk snapshot (afa7b78, March 4, 2016)
- Release build of JerryScript
- Optimizing for the Cortex-M4F
- Building with -Os
- Static build, running bare-metal on the STM32F4 board

# Code size


-Os -mthumb -mcpu=cortex-m4 -mfpu=fpv4-sp-d16
Code size in KB (lower is better)
250
200
150
100
50
0
231
208
190
JerryScript
Clang r262611
GCC 5.2.1
GCC 5.2.1 LTO

# Code size - LTO

- GCC in LTO mode produces the smallest binary
- Clang has no support for building in -Os -flto
- LTO optimization pipeline not tuned for size

# Identifying interesting functions

- Python script which parses the "objdump -d" output
- Parsing the assembly dump of both binaries
- Binaries compiled with -fno-inline
- Computes per function code size delta
- Some functions get specialized and are only available in one binary
- Not perfectly accurate but good enough

# Code size

```
/**
 * Creates and returns a jerry_api_value_t
 * with type JERRY_API_DATA_TYPE_NULL.
 */
jerry_api_value_t
jerry_api_create_null_value (void)
{
  jerry_api_value_t jerry_val;
  jerry_val.type = JERRY_API_DATA_TYPE_NULL;
  return jerry_val;
}
```

```
typedef struct jerry_api_value_t
{
  jerry_api_data_type_t type;

  union
  {
    bool v_bool;
    double v_float64;
    uint32_t v_uint32;
    ...
  } u;
} jerry_api_value_t;
```

```
Clang
<jerry_api_create_null_value>:
b083            sub     sp, #12
2102            movs    r1, #2
f840 1b04       str.w   r1, [r0], #4
4669            mov     r1, sp
e891 100c       ldmia.w r1, {r2, r3, ip}
e880 100c       stmia.w r0, {r2, r3, ip}
b003            add     sp, #12
4770            bx      lr
```

```
GCC
<jerry_api_create_null_value>:
2302            movs    r3, #2
7003            strb    r3, [r0, #0]
4770            bx      lr
```

# Code size

```
void
lit_magic_strings_ex_set (const lit_utf8_byte_t **ex_str_items,
                          uint32_t count,
                          const lit_utf8_size_t *ex_str_sizes)
{
  lit_magic_string_ex_array = ex_str_items;
  lit_magic_string_ex_count = count;
  lit_magic_string_ex_sizes = ex_str_sizes;
}
```

```
Clang
<lit_magic_strings_ex_set>:
f240 0340       movw    r3, #64 ; 0x40
f2c2 0300       movt    r3, #8192       ; 0x2000
6018            str     r0, [r3, #0]
f240 0044       movw    r0, #68 ; 0x44
f2c2 0000       movt    r0, #8192       ; 0x2000
6001            str     r1, [r0, #0]
f240 0048       movw    r0, #72 ; 0x48
f2c2 0000       movt    r0, #8192       ; 0x2000
6002            str     r2, [r0, #0]
4770            bx      lr
```

```
GCC
<lit_magic_strings_ex_set>:
      4b01            ldr     r3, [pc, #4]    ; (base)
      e883 0007       stmia.w r3, {r0, r1, r2}
      4770            bx      lr
base: 20000074        .word   0x20000074
```

# Code size

```
/* Decompress extended compressed pointer. */
lit_record_t *
lit_cpointer_decompress (lit_cpointer_t compressed_pointer)
{
  if (compressed_pointer == MEM_CP_NULL)
  {
    return NULL;
  }

  return (lit_record_t *) mem_decompress_pointer (compressed_pointer);
}
```

```
Clang
<lit_cpointer_decompress>:
         b120            cbz     r0, 8001de2 <lit_cpointer_decompress+0xc>
         b580            push    {r7, lr}
         466f            mov     r7, sp
         f000 fba2       bl      8002524 <mem_decompress_pointer>
         bd80            pop     {r7, pc}
8001de2: 2000            movs    r0, #0
         4770            bx      lr
```

```
GCC
<lit_cpointer_decompress>:
         b108            cbz     r0, 8001a74 <lit_cpointer_decompress+0x6>
         f000 ba6a       b.w     8001f48 <mem_decompress_pointer>
8001a74: 4770            bx      lr
```

# Code size

```
/* Common function to generate fixed width, right aligned decimal numbers. */
static lit_utf8_byte_t *
ecma_date_value_number_to_bytes (lit_utf8_byte_t *dest_p,
                                 int32_t number,
                                 int32_t width)
{
  dest_p += width;
  lit_utf8_byte_t *result_p = dest_p;

  do
  {
    dest_p--;
    *dest_p = (lit_utf8_byte_t) ((number % 10) + (int32_t) LIT_CHAR_0);
    number /= 10;
  }
  while (--width > 0);

  return result_p;
}
```

# Code size

```
/* Common function to generate fixed width, right aligned decimal numbers. */
static lit_utf8_byte_t *
ecma_date_value_number_to_bytes (lit_utf8_byte_t *dest_p,
                                 int32_t number,
                                 int32_t width) {
  dest_p += width;
  lit_utf8_byte_t *result_p = dest_p;
  do
  {
    dest_p--;
    *dest_p = (lit_utf8_byte_t) ((number % 10) + (int32_t) LIT_CHAR_0);
    number /= 10;
  }
  while (--width > 0);
  return result_p;
}
```

```
Clang
<ecma_date_value_number_to_bytes>:
      b510       push    {r4, lr}
      4686       mov     lr, r0
      f246 6c67  movw    ip, 0x6667
      eb0e 0002  add.w   r0, lr, r2
      3a01       subs    r2, #1
      f2c6 6c66  movt    ip, 0x6666

loop: fb51 f30c  smmul   r3, r1, ip
      109c       asrs    r4, r3, #2
      eb04 73d3  add.w   r3, r4, r3, lsr #31
      eb03 0483  add.w   r4, r3, r3, lsl #2
      eba1 0144  sub.w   r1, r1, r4, lsl #1
      3130       adds    r1, #48 ; 0x30
      f80e 1002  strb.w  r1, [lr, r2]
      1e51       subs    r1, r2, #1
      3201       adds    r2, #1
      2a01       cmp     r2, #1
      460a       mov     r2, r1
      4619       mov     r1, r3

      dced       bgt.n   loop
      bd10       pop     {r4, pc}
      bf00       nop
```

```
GCC
<ecma_date_value_number_to_bytes>:
      4410       add     r0, r2
      b530       push    {r4, r5, lr}
      4603       mov     r3, r0
      250a       movs    r5, #10
      1a12       subs    r2, r2, r0

loop: fb91 f4f5  sdiv    r4, r1, r5
      fb05 1114  mls     r1, r5, r4, r1
      3130       adds    r1, #48 ; 0x30
      f803 1d01  strb.w  r1, [r3, #-1]!
      4621       mov     r1, r4
      189c       adds    r4, r3, r2
      2c00       cmp     r4, #0

      dcf4       bgt.n   loop
      bd30       pop     {r4, r5, pc}
```

# Code size

```
ecma_string_t *
ecma_get_string_from_value (ecma_value_t value) {
  JERRY_ASSERT (ecma_get_value_type_field (value) == ECMA_TYPE_STRING);

  return ECMA_GET_NON_NULL_POINTER (ecma_string_t,
                                    ecma_get_value_value_field (value));
}

ecma_object_t *
ecma_get_object_from_value (ecma_value_t value) {
  JERRY_ASSERT (ecma_get_value_type_field (value) == ECMA_TYPE_OBJECT);

  return ECMA_GET_NON_NULL_POINTER (ecma_object_t,
                                    ecma_get_value_value_field (value));
}
```

```
Clang
<ecma_get_object_from_value>:
b580      push    {r7, lr}
466f      mov     r7, sp
f7ff ff28 bl <ecma_get_value_value_field>
e8bd 4080 ldmia.w sp!, {r7, lr}
f7f5 bad2 b.w <mem_decompress_pointer>

<ecma_get_string_from_value>:
b580      push    {r7, lr}
466f      mov     r7, sp
f7ff ff30 bl <ecma_get_value_value_field>
e8bd 4080 ldmia.w sp!, {r7, lr}
f7f5 bada b.w <mem_decompress_pointer>
```

```
GCC
<ecma_get_object_from_value>:
b508      push {r3, lr}
f7ff ff3c bl <ecma_get_value_value_field>
e8bd 4008 ldmia.w sp!, {r3, lr}
f7f5 bfd1 b.w <mem_decompress_pointer>

<ecma_get_string_from_value>:
f7ff bff7 b.w <ecma_get_object_from_value>
```

# Code size

```
bool
lit_literal_equal_type_utf8 (lit_literal_t lit,
                             const lit_utf8_byte_t *str_p,
                             lit_utf8_size_t str_size)
{
  const lit_record_type_t type = (const lit_record_type_t) lit->type;

  if (type == LIT_RECORD_TYPE_NUMBER || type == LIT_RECORD_TYPE_FREE)
  {
    return false;
  }

  return lit_literal_equal_utf8 (lit, str_p, str_size);
}
```

Clang

```
<lit_literal_equal_type_utf8>:
7883      ldrb    r3, [r0, #2]
f043 0304 orr.w   r3, r3, #4
b2db      uxtb    r3, r3
2b04      cmp     r3, #4
bf04      itt     eq
2000      moveq   r0, #0
4770      bxeq    lr
b580      push    {r7, lr}
466f      mov     r7, sp
f7ff ff17 bl      8001fac <lit_literal_equal_utf8>
bd80      pop     {r7, pc}
```

GCC

```
<lit_literal_equal_type_utf8>:
      7883      ldrb    r3, [r0, #2]
      f013 03fb ands.w  r3, r3, #251    ; 0xfb
      d001      beq.n   exit
      f7ff bfa6 b.w     8001c3a <lit_literal_equal_utf8>
exit: 4618      mov     r0, r3
      4770      bx      lr
```

# Code size

```
void libc_fatal (const char *msg,
                 const char *file_name,
                 const char *function_name,
                 const unsigned int line_number) {
  if (msg != NULL
      && file_name != NULL
      && function_name != NULL)
  {
    printf ("Assertion '%s' failed at %s (%s:%u).\n",
            msg, function_name, file_name, line_number);
  }
  abort ();
}
```

```
Clang
<libc_fatal>:
    b580      push    {r7, lr}
    466f      mov     r7, sp
    b082      sub     sp, #8
    468c      mov     ip, r1
    4601      mov     r1, r0
    b161      cbz     r1, L1
    f1bc 0f00 cmp.w   ip, #0
    bf18      it      ne
    2a00      cmpne   r2, #0
    d007      beq.n   L1
    f24a 405d movw    r0, #42077
    9300      str     r3, [sp, #0]
    4663      mov     r3, ip
    f6c0 0003 movt    r0, #2051
    f7ff fee4 bl      802d57e <printf>
L1: f000 f804 bl      802d7c2 <abort>
```

```
GCC
<libc_fatal>:
    b507      push    {r0, r1, r2, lr}
    b138      cbz     r0, L1
    b131      cbz     r1, L1
    b12a      cbz     r2, L1
    9300      str     r3, [sp, #0]
    460b      mov     r3, r1
    4601      mov     r1, r0
    4802      ldr     r0, [pc, #8] ; L2
    f7ff fe74 bl      802855c <printf>
L1: f000 f805 bl      8028882 <abort>
L2: 0803515a  .word   0x0803515a
```

# Code size

```
bool
lit_char_is_hex_digit (ecma_char_t c)
{
  return ((c >= '0' && c <= '9')
          || (c >= 'a' && c <= 'f')
          || (c >= 'A' && c <= 'F'));
}
```

```
Clang
<lit_char_is_hex_digit>:
    f1a0 0130 sub.w   r1, r0, #48 ; 0x30
    b289      uxth    r1, r1
    290a      cmp     r1, #10
    bf22      ittt    cs
    f1a0 0161 subcs.w r1, r0, #97 ; 0x61
    b289      uxthcs  r1, r1
    2906      cmpcs   r1, #6
    d306      bcc.n   L1
    3841      subs    r0, #65 ; 0x41
    b281      uxth    r1, r0
    2000      movs    r0, #0
    2906      cmp     r1, #6
    bf38      it      cc
    2001      movcc   r0, #1
    4770      bx      lr
L1: 2001      movs    r0, #1
    4770      bx      lr
```

```
GCC
<lit_char_is_hex_digit>:
    f1a0 0330 sub.w   r3, r0, #48 ; 0x30
    2b09      cmp     r3, #9
    d907      bls.n   L1
    f020 0020 bic.w   r0, r0, #32
    3841      subs    r0, #65 ; 0x41
    2805      cmp     r0, #5
    bf8c      ite     hi
    2000      movhi   r0, #0
    2001      movls   r0, #1
    4770      bx      lr
L1: 2001      movs    r0, #1
    4770      bx      lr
```

# Recap

- Recap: List of potential optimizations
  - Constant pool vs. materialization
  - CBZ; more efficient branching
  - Same code merging
  - Switch lowering
  - Tail calls
  - Data packing
  - GCC-generated computations often more compact
  - LLVM frequently emits the IT instruction

# Summary

# Summary

- Clang-generated code about 10% bigger
- Lots of code size-related low-hanging fruit
- Future work: Enable LTO + -Os in Clang
- Specific optimization passes for code size

# Thank you.