Code Size Optimisations for ARM

ARM

James Molloy, Sjoerd Meijer, Pablo Barrio, Kristof Beyls

EuroLLVM'17

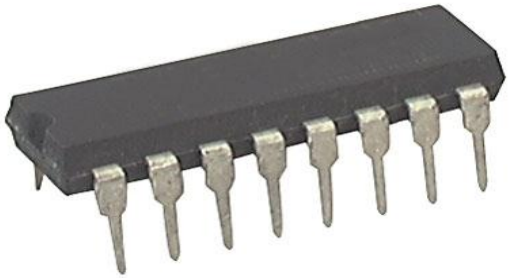
March 2017

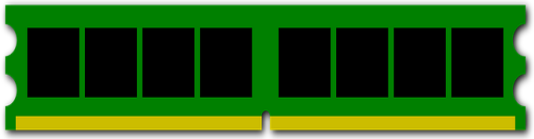
Code size…

"…is good, until it isn't anymore
(all of a sudden)"

Code size matters

- Not uncommon for a micro-controller to have:
  - 64 Kbytes of Flash
  - 8 Kbytes of RAM





- Show stopper for many embedded systems and applications!

Problem statement

- Very costly when images don't fit in RAM or ROM
  - Bigger memories,
  - More power hungry,
  - HW redesign,
  - And more…
- Code size optimisations are crucial
- We found that LLVM's code generation not good enough when optimising for size.

Idioms in embedded code

- Dominated by a lot of control flow decisions based on peripheral register states:
  - control code (if-statements, switches),
  - magic constants,
  - bitwise operations:

```
if ((((StructA*) (((uint32_t)0x40000000) + 0x6400))->M1
     & ((uint32_t)0x00000002)) != 0U) {
  …
```

  - Filling in data structures:

```
ptr->structarr[idx].field3 &= ~((uint32_t)0x0000000F);
```

Implemented improvements

Summary of improvements

- About 200 patches and contributions last year (all upstream)
  - Touched many different parts in both the middle-end and backend.
- Categorise them in these 4 areas:
  1. turn off specific optimisations when optimising for size
  2. tuning optimisations,
  3. constants,
  4. bit twiddling.
- Target independent: 1- 3, target dependent: 4
- Target Thumb code (and **not** e.g. AArch64)
  - Provides 32-bit and 16-bit instruction encodings

Category 1: turn off specific optimisations

- Code size more valuable than execution time in this market
- Patch 1:

```
case LibFunc::fputs:
     if (optForSize())
       return nullptr;
      return optimizeFPuts(CI, Builder);
```

- Patch 2:

```
// when optimising for size, we don't want to
// expand a div to a mul and a shift.
if (ForCodeSize)
   return SDValue();
```

Category 1: turn off specific optimisations

- Some other commits:
  - do not inline memcpy if expansion is bigger than the lib call.
  - Machine Block Placement: do not reorder and move up loop latch block to avoid extra branching
  - Do not expand UDIV/SDIV to multiplication sequence.

- In summary:
  - Bunch of simple patches to turn off performance optimisations that increase code size
  - Optimisations/transformations focus on performance
    - It wasn't really bad; a lot of passes do check the optimisation level,
    - But clearly not enough!

Category 2: tuning optimisations

- SimplifyCFG:
  - Performs dead code elimination,
  - basic block merging (chain of blocks with 1 predecessor/successor)
  - adjusts branches to branches
  - Eliminate blocks with just one unconditional branch

- And also "one stop shop for all your CFG peephole optimisations":
  - Hoist conditional stores
  - Merge conditional stores
  - **Range reduce switches**
  - **Sink common instructions down to the end block**

Category 2: tuning transformations - SimplifyCFG

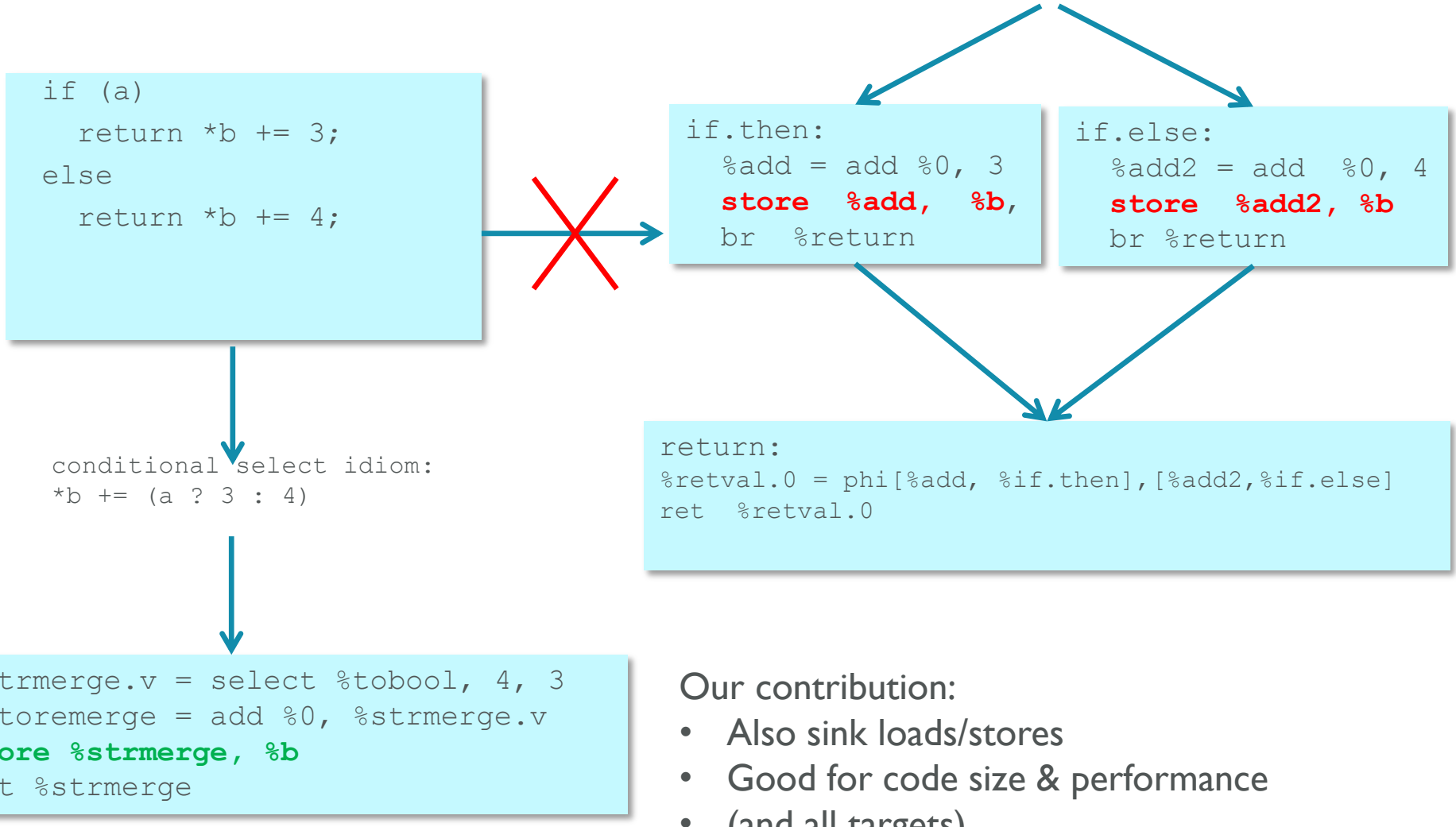
- Rewrite sparse switches to dense switches:

```
switch (i) {
 case 5: ...
 case 9: ...
 case 13: ...
 case 17: ...
}
```

```
if ( (i - 5) % 4 ) goto default;
switch ((i - 5) / 4) {
   case 0: ...
   case 1: ...
   case 2: ...
   case 3: ... }
```

- Real life example: switching over memory addresses
- Dense switches can be lowered better (not our contribution):
  - E.g. transformed into lookup tables
  - Good for code size & performance

Category 2: tuning transformations



Our contribution:

- Also sink loads/stores
- Good for code size & performance
- (and all targets)

ARM

Category 2: tuning transformations

- Some other commits:
  - Inlining heuristics have been adapted
  - Jump threading: unfold selects that depend on the same condition
  - tailcall optimization: relax restriction on variadic functions
- Instruction selection:
  - Lower UDIV+UREM more efficiently (not use libcalls)
  - Lower pattern of certain selects to SSAT

Category 3: constants

Immediate offsets available on store instructions:

Instruction	Imm. offset
32-bit encoding, word, halfword, or byte	–255 to 4095
32-bit encoding, doubleword	–1020 to 1020
16-bit encoding, word	0 to 124
16-bit encoding, halfword	0 to 62
16-bit encoding, byte	0 to 31
16-bit encoding, word, Rn is SP	0 to 1020

- Strategy is to use narrower instructions
- More constrained
- Accurate analysis required

https://developer.arm.com/docs/dui0801/f/a32-and-t32-instructions/str-immediate-offset

Category 3: constant hoisting

- From a set of constants in a function:
  - pick constant with most uses,
  - Other constants become an offset to that selected base constant.

Constants	2	4	12	44
NumUses	3	2	**8**	7

- Selecting 12 as the base constant:

Imm Offset	-10	-8	0	32

- 12 stores with 4 byte encoding, 8 stores with 2 byte encoding (when the range is 0..31)

Category 3: constant hoisting

- Objective: maximise the constants in range:

Constants	2	4	12	44
NumUses	3	2	**8**	7

- Now we select 2 as the base constant:

Imm. Offset	0	2	10	42
NumUses	**3**	**2**	**8**	7

- 7 stores with 4-byte encoding, 13 stores with 2-byte encoding
- Code size reduction of (13 – 8) * 2 = 10 bytes.

Category 3: constants

- For transformations, it's crucial to use and have accurate cost models
- For constants, this is provided by TargetTransformInfo
  - Query properties, sizes, costs of immediates

- Some other commits tweaked/added:
  - TTI::getIntImmCodeSizeCost();
  - TTI::getIntImmCost()

- And another commit:
  - Promotes small global constants to constant pools

Category 4: bit twiddling

- A branch on a compare with zero:

  (CMPZ (AND x, #bitmask), #0)

- CMPZ is a compare that sets only Z flag in LLVM
- Can be replaced with 1 instruction (most of the time). But how?

AND	r0, r0, #3	4 bytes
CMP	r0, #0	2 bytes
BEQ	.LBB0_2	2 bytes
		8 bytes

Category 4: bit twiddling, cont'd

- The ALU status flags:
  - **N**: set when the result of the operation was **N**egative.
  - **Z**: set when the result of the operation was **Z**ero.
  - **C**: set when the operation resulted in a **C**arry.
  - **V**: Set when the operation caused o**V**erflow.

Flag setting ANDS:
Don't need the CMP:

ANDS	r0, #3	4 bytes
BEQ	.LBB0_2	2 bytes

6 bytes

If bitmask is consecutive seq. of bits,
And if it touches the LSB,
Remove all upper bits:

LSLS	r0, r0, #30	2 bytes
BEQ	.LBB0_2	2 bytes

4 bytes

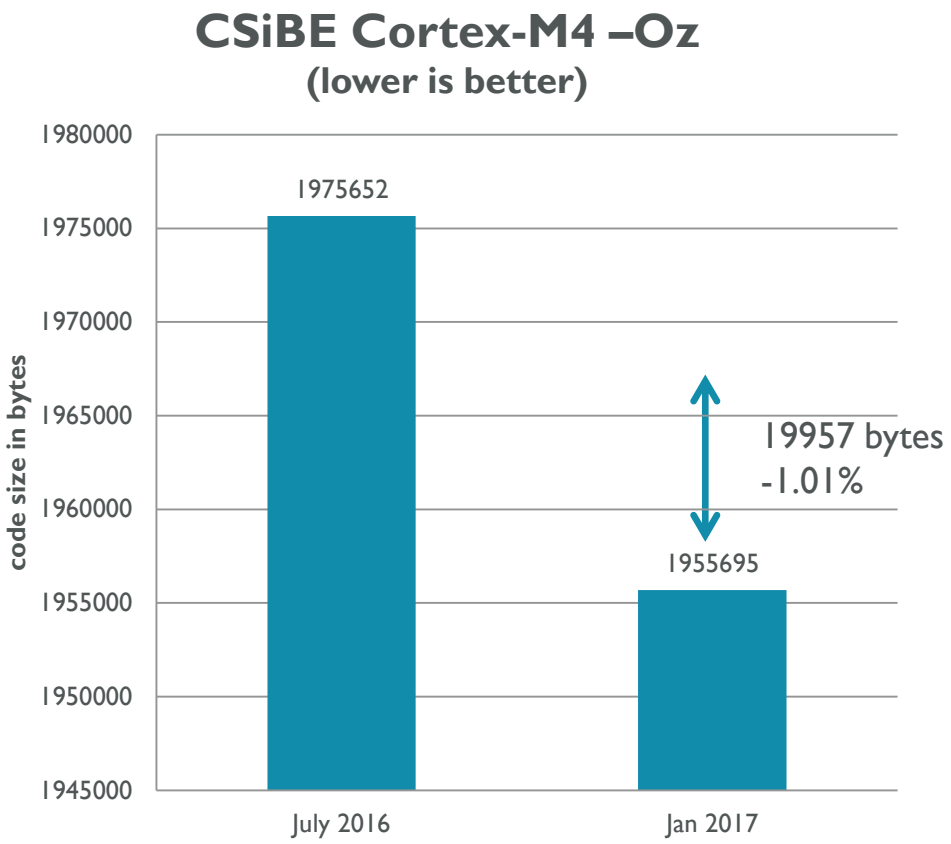
Category 4: bit twiddling, cont'd

- Some more commits:
  - Remove CMPs when we care only about the N and Z flags
  - A CMP with -1 can be done by adding 1 and comparing against 0
- Summary:
  - There are many, many tricks (see also Hacker's Delight)
  - Although mostly small rewrites, they can give good savings if there are lot of them.

Experimental results

Results CSiBE-v2.1.1

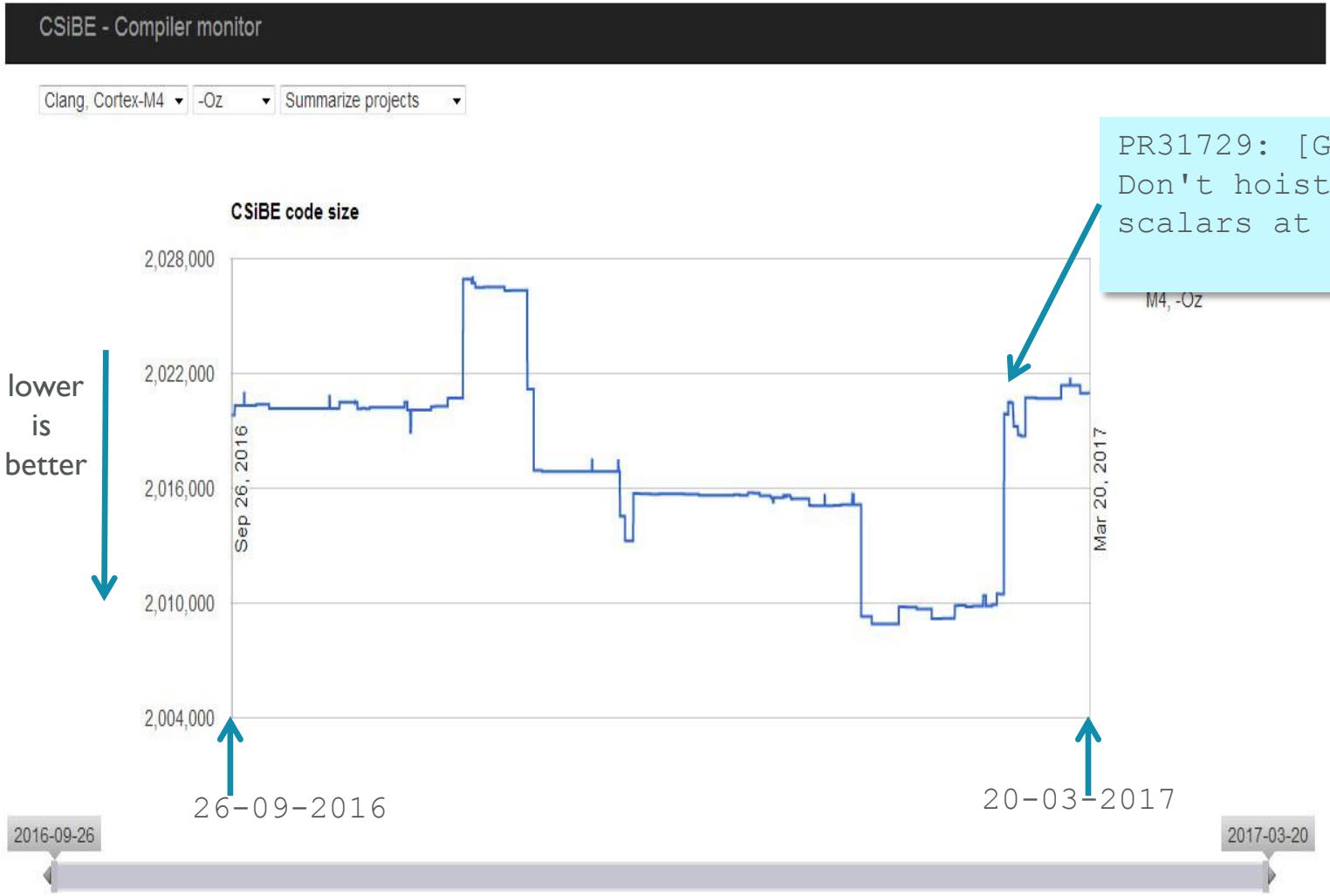
- CSiBE: code size benchmark
  - http://szeged.github.io/csibe/
  - Jpeg, flex, lwip, OpenTCP, replaypc Libpng, libmspack, zlib,
- Setup:
  - -Oz -mcpu=cortex-m4 -mthumb
  - Includes our contributions,
  - but everyone else's too!



Improvements: 337, Unchanged: 154, Regressions: 127

ARM

CSiBE: Cortex-M4, -Oz



http://szeged.github.io/csibe/compiler-monitor.html

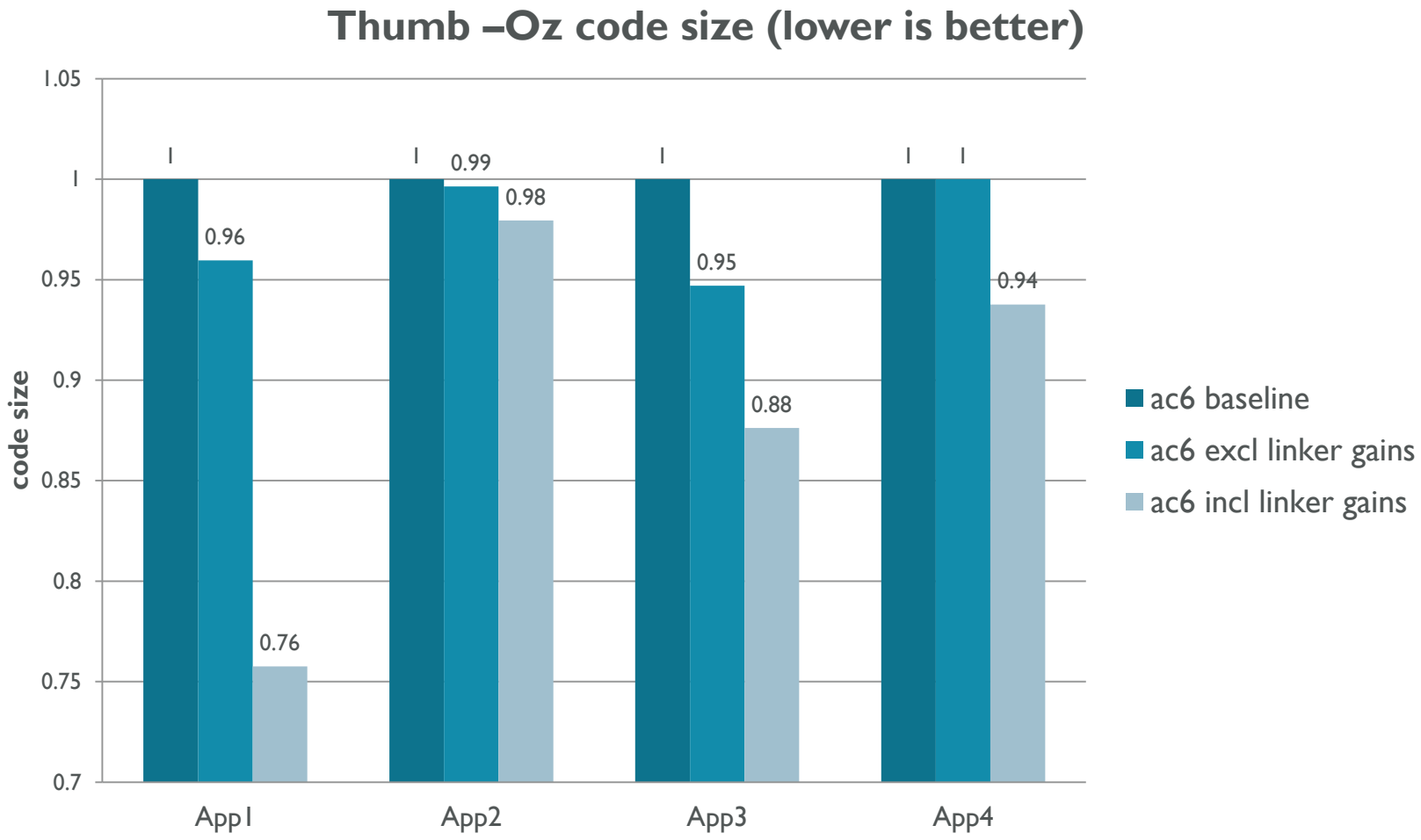
ARM

More results

- ARM Compiler 6 toolchain
  - LLVM based compiler
  - Proprietary linker, and libraries*
- Code generation is only part of the puzzle:
  - Library selection:
    - Different library variants with e.g. different IEEE math lib compliance
  - Linker can e.g.:
    - Remove unused sections,
    - Partially include libraries.

*** ARM would welcome lld picking up the challenge of producing really good, compact code, and ARM would help.**

ARM Compiler 6 Results



ARM

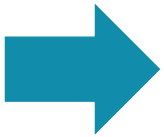
Further potential improvements

Future work

- Avoid wide branches
- Spilling of small constants
  - balance materialization and register pressure
- Constant hoisting too aggressive

Future work: Machine Block Placement

```
int foo(int *p, int *q)
{
   if (!p) return ERR;
   if (!q) return ERR;
   ..
   if (..) return ERR;
   ..
   // lot of code here
   ..
   return SUCC;
}
```



```
BB0:
    ..
    cbz  r0, .LBB0_3

BB1:
    ..
    cbz  r4, .LBB0_3

BB342:
// lot of code here

.LBB0_3:
    mov.w    r0, #-1
    pop
```

- Wide branches to exit block(s)
- MPB: should take into account branch distances (for code size)

Future work: Constant Hoisting

```
Entry:
  movs   r2, #1
  lsls   r3, r2, #15
  lsls   r0, r2, #19
  str    r0, [sp, #8]    @ 4-byte Spill
  lsls   r0, r2, #20
  str    r0, [sp, #12]   @ 4-byte Spill
  lsls   r0, r2, #21
  str    r0, [sp, #16]   @ 4-byte Spill
  lsls   r0, r2, #22
  str    r0, [sp]        @ 4-byte Spill
  lsls   r6, r2, #25
  movs   r0, #3
  ...
```

- Constant hoisting is really aggressive
- Does not take into account register pressure

Future work:
Balance materialization and register pressure

Save #2 into a stack slot:

```
movs    r6,#2
mov     r0,r6
blx     r1
cmp     r0,#0
bne     {pc}+0xfa
str     r6,[sp,#0x10]
```

- Rematerialization: clone of an instruction where it is used
  - Cannot have have any sideeffects
  - In thumb-1, MOVS always sets the flags
- Hoist constants to avoid the materialization vs. trying to sink them to reduce register pressure

Conclusions

- Good code size improvements:
  - Open Source LLVM: CSiBE-v2.1.1 improved by1.01%
  - ARM Compiler:
    - From 1% to 6% across a range of microcontroller applications (code generation)
    - From 2% to 24% fully using the ARM Compiler toolchain (armlink)
  - widely applicable to a lot of code
- Achieved a lot in relatively short amount of time.
  - Shows LLVM is not in a bad place!
- There's (always) more to do:
  - Focussed on 4 realistic microcontroller application examples
  - Picked a lot (most?) of low hanging fruit, and also did a few big tasks
  - But we have left a few big tasks on the table.

ARM