Code Size Optimisations for 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:

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

- Filling in data structures:

```c
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:

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

- Patch 2:

```c
// 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:

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

```c
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

if (a)
    return \*b += 3;
else
    return \*b += 4;

conditional select idiom:
\*b += (a ? 3 : 4)

%strmerge.v = select %tobool, 4, 3
%storemerge = add %0, %strmerge.v
store %strmerge, \%b
ret %strmerge

Our contribution:
• Also sink loads/stores
• Good for code size & performance
• (and all targets)
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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Imm. offset</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>32-bit</strong> encoding, word, halfword, or byte</td>
<td>−255 to 4095</td>
</tr>
<tr>
<td><strong>32-bit</strong> encoding, doubleword</td>
<td>−1020 to 1020</td>
</tr>
<tr>
<td>16-bit encoding, word</td>
<td>0 to 124</td>
</tr>
<tr>
<td>16-bit encoding, halfword</td>
<td>0 to 62</td>
</tr>
<tr>
<td>16-bit encoding, byte</td>
<td>0 to 31</td>
</tr>
<tr>
<td>16-bit encoding, word, Rn is SP</td>
<td>0 to 1020</td>
</tr>
</tbody>
</table>

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

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.

<table>
<thead>
<tr>
<th>Constants</th>
<th>2</th>
<th>4</th>
<th>12</th>
<th>44</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumUses</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

- 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:

<table>
<thead>
<tr>
<th>Constants</th>
<th>2</th>
<th>4</th>
<th>12</th>
<th>44</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumUses</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

- Now we select 2 as the base constant:

<table>
<thead>
<tr>
<th>Imm. Offset</th>
<th>0</th>
<th>2</th>
<th>10</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumUses</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

- 7 stores with 4-byte encoding, 13 stores with 2-byte encoding
- Code size reduction of \((13 - 8) \times 2 = 10\) 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:

  \[(\text{CMPZ} \ (\text{AND} \ x, \ #\text{bitmask}), \ #0)\]

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>R0, R0, #3</th>
<th>4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>r0, #0</td>
<td>2 bytes</td>
</tr>
<tr>
<td>CMP</td>
<td>.LBB0_2</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

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**verflow.

#### Flag setting ANDS:

<table>
<thead>
<tr>
<th>ANDS</th>
<th>r0, #3</th>
<th>4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>.LBB0_2</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>LSLS</th>
<th>r0, r0, #30</th>
<th>2 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>.LBB0_2</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

6 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
  - Jpeg, flex, lwip, OpenTCP, replaypc
  - Libpng, libmspack, zlib,

- Setup:
  - `-Oz -mcpu=cortex-m4 -mthumb`
  - Includes our contributions,
  - but everyone else’s too!

---

CSiBE Cortex-M4 –Oz
(lower is better)

Improvements: 337, Unchanged: 154, Regressions: 127
CSiBE: Cortex-M4, -Oz

PR31729: [GVNHoist]
Don't hoist unsafe scalars at -Oz

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

Thumb –Oz code size (lower is better)

- ac6 baseline
- ac6 excl linker gains
- ac6 incl linker gains

<table>
<thead>
<tr>
<th>App</th>
<th>ac6 baseline</th>
<th>ac6 excl linker gains</th>
<th>ac6 incl linker gains</th>
</tr>
</thead>
<tbody>
<tr>
<td>App1</td>
<td>0.96</td>
<td>0.76</td>
<td></td>
</tr>
<tr>
<td>App2</td>
<td>0.99</td>
<td>0.98</td>
<td></td>
</tr>
<tr>
<td>App3</td>
<td>0.95</td>
<td>0.88</td>
<td></td>
</tr>
<tr>
<td>App4</td>
<td>0.94</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

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

- 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:

<table>
<thead>
<tr>
<th>movs</th>
<th>r6, #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>r0, r6</td>
</tr>
<tr>
<td>blx</td>
<td>r1</td>
</tr>
<tr>
<td>cmp</td>
<td>r0, #0</td>
</tr>
<tr>
<td>bne</td>
<td>{pc} + 0xfa</td>
</tr>
<tr>
<td>str</td>
<td>r6, [sp, #0x10]</td>
</tr>
</tbody>
</table>

- Rematerialization: clone of an instruction where it is used
  - Cannot have any side effects
  - 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 by 1.01%
  - 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.