Code size Improvement work in TCWG

Peter Smith, Oliver Stannard and Yvan Roux
Topics covered

- Investigation in embedded code size GCC vs Clang
- Virtual Function Elimination
- Function Outlining support in Arm
TCWG Code Size Investigation

- Compare Clang and GCC for code-size on embedded systems
- Traditional territory of GNU ARM Embedded Toolchain for Cortex-M and Cortex-R processors
- Compare Clang master (10.0.0) and GCC Trunk (10.0.0)
- Focus on Clang -Oz and GCC -Os for minimum code-size using comparable options
- Primary goal to find out how close Clang is to GCC on code-size
- Secondary goal to record opportunities for improvements
- Choose representative real-world code-bases rather than benchmarks
  - Zephyr (RTOS) for cortex-m4 soft-float and cortex-m0
  - CMSIS-DSP for cortex-m4f
Code size comparison with Zephyr

- Measure code size for a subset of the Zephyr sample applications
- For GCC, use a TCWG built arm-none-eabi-gcc with GNU-RM multilibs
- For Clang, use a TCWG build clang installation using arm-none-eabi-gcc as the linker driver
  - Same libraries and linker, Clang used as assembler and compiler
  - Command line options chosen to match arm-none-eabi-gcc defaults
GCC and Clang code size comparison

- Measure the size of the executable where possible
  - Comdat groups will be merged by the linker
  - Executable will include helper library code
  - Effect of linker garbage collection on dead code
- Make sure the same binary libraries are used in any comparison
- The size of the .text section is a good proxy for overall code size
- The ELF symbol is a good measurement for a function's code size
Cortex-M4 results normalized to GCC

- Normalizing GCC at 1.0 shows the relative differences more clearly
- Graph y-range is compressed to emphasise differences
- Total LLVM Size 3462456
- Total GCC Size 3424092
- LLVM is 1.1% larger
Cortex-M0 results normalized to GCC

- More samples show larger increases
- Graph y-range is compressed to emphasise differences.
- Cortex-M0 has fewer supported samples.
- Total LLVM Size 2474544
- Total GCC Size 2404132
- LLVM is 2.8% larger
Analysis of results

- On Cortex M4 LLVM is about 1% larger
- On Cortex M0 LLVM is about 2.8% larger
- Amount of C-library and helper library code < 1%
- Common code to all examples about 6000 bytes on M4; 5% of largest example, 60% of hello_world
- Turning off inlining halves the difference
- Fewer spills due to register allocation in GCC
- Many smaller optimizations in aggregate
CMSIS DSP

- Part of the CMSIS library
- Implementations of DSP functions
- Structure of library amenable to examining object files
CMSIS DSP Cortex-M4f normalized to GCC

LLVM normalized to GCC

- Clang is 90% the size of GCC for CMSIS DSP on M4
Analysis of results

- Clang can be substantially smaller than GCC
- Very hard for a non-expert to compare the generated code
- Many smaller things, but can be repeated due to nature of the code
  - GCC inlines aeabi_llsl in several places.
  - Clang makes use of rounding instructions such as SMMULR for source that manually rounds.
    - \((q31_t) (((((q63_t) a) \ll 32) + ((q63_t) x \times y) + 0x80000000LL) \gg 32)\)
Summary of Code Size investigation

- GCC is on average smaller than Clang for Zephyr for both Cortex-M0 and Cortex-M4
  - Inlining strategy the largest single difference
  - Many small improvements needed
- Cortex-M4 closer (1%) than Cortex-M0 (2.9%)
- Clang generates smaller DSP code for Cortex-M4 in almost all cases
- Specific issues recorded in LLVM-583
Dead Virtual Function Elimination (VFE) in LLVM

Oliver Stannard
An example we’d like to optimise

```cpp
struct A {
    virtual int foo() { return 1; }
    virtual int bar() { return 2; }
};

struct B : A {
    virtual int foo() { return 3; }
    virtual int bar() { return 4; }
};

A* make_A() { return new A(); }
B* make_B() { return new B(); }

int call_1(A* p) { return p->foo(); }
int call_2(B* p) { return p->bar(); }
```

No calls to this function

Could call A::foo or B::foo

Can only call B::bar
What does this look like to GlobalDCE?

Other code

A's constructor

A's VTable

A::foo

A::bar

B's constructor

B's VTable

B::foo

B::bar

Call site 1

Call site 2

- Load vtable ptr from object
- Load func ptr from vtable
- Call func using ptr
Type metadata

- Optionally added to vtables, virtual call sites
- Provides link between call sites and vtable slots
- Currently used for devirtualisation, control flow integrity
- Needs some changes to call sites to allow VFE

```llvm
@_ZTV1A = constant { [4 x i8*] } ..., !type !0
@_ZTV1B = constant { [4 x i8*] } ..., !type !0, !type !4

!0 = !{i64 16, !"_ZTS1A"}
!4 = !{i64 16, !"_ZTS1B"}

%vtable1 = load i8*, i8** %0, align 8, !tbaa !9
%1 = tail call { i8*, i1 } @llvm.type.checked.load(
                   i8* %vtable1, i32 8, metadata !"_ZTS1B")
```
How does this change GlobalDCE’s view?

- Other code
- A’s constructor
- B’s constructor
- Call site 1
- Call site 2
- A’s VTable
- B’s VTable
- A::foo
- A::bar
- B::foo
- B::bar
When is this optimisation valid?

- Need to see every possible call site involving a class
- Generally requires `-fvisibility=hidden`, and LTO
- More complicated if related classes have different visibility
  - Most visible base class with a vtable
- Added new `!vcall_visibility` metadata to represent this
Benchmark - SPEC2006 (C++ subset)

Code size change, compared to -Oz -flto -fvisibility=hidden
Benchmark - mbed-os examples

Code size change, compared to -Oz -flto -fvisibility=hidden
Future work

- Commit the patch (in review)
- Turn on by default
- ThinLTO - currently requires full LTO
- Dead RTTI elimination
Machine Outliner for Arm

Yvan Roux
LLVM Machine Outliner

- Code-size reduction pass
- Replacing repeated sequences of code by calls to equivalent functions
- Introduced by Apple for AArch64 and X86 in 2017
Example

foo:

```assembly
movw r0, :lower16:a
movt r0, :upper16:a
ldr r0, [r0]
movw r1, :lower16:b
movt r1, :upper16:b
ldr r1, [r1]
add r0, r0, r1
movw r1, :lower16:c
movt r1, :upper16:c
ldr r1, [r1]
add r0, r0, r1
movw r1, :lower16:d
movt r1, :upper16:d
ldr r1, [r1]
add r0, r0, r1
bx lr
```

bar:

```assembly
movw r0, :lower16:a
movt r0, :upper16:a
ldr r0, [r0]
movw r1, :lower16:b
movt r1, :upper16:b
ldr r1, [r1]
add r0, r0, r1
movw r1, :lower16:c
movt r1, :upper16:c
ldr r1, [r1]
add r0, r0, r1
movw r1, :lower16:d
movt r1, :upper16:d
ldr r1, [r1]
sub r0, r0, r1
bx lr
```
### Example

<table>
<thead>
<tr>
<th>foo:</th>
<th>bar:</th>
</tr>
</thead>
<tbody>
<tr>
<td>movw r0, :lower16:a</td>
<td>movw r0, :lower16:a</td>
</tr>
<tr>
<td>movt r0, :upper16:a</td>
<td>movt r0, :upper16:a</td>
</tr>
<tr>
<td>ldr r0, [r0]</td>
<td>ldr r0, [r0]</td>
</tr>
<tr>
<td>movw r1, :lower16:b</td>
<td>movw r1, :lower16:b</td>
</tr>
<tr>
<td>movt r1, :upper16:b</td>
<td>movt r1, :upper16:b</td>
</tr>
<tr>
<td>ldr r1, [r1]</td>
<td>ldr r1, [r1]</td>
</tr>
<tr>
<td>add r0, r0, r1</td>
<td>add r0, r0, r1</td>
</tr>
<tr>
<td>movw r1, :lower16:c</td>
<td>movw r1, :lower16:c</td>
</tr>
<tr>
<td>movt r1, :upper16:c</td>
<td>movt r1, :upper16:c</td>
</tr>
<tr>
<td>ldr r1, [r1]</td>
<td>ldr r1, [r1]</td>
</tr>
<tr>
<td>add r0, r0, r1</td>
<td>add r0, r0, r1</td>
</tr>
<tr>
<td>movw r1, :lower16:d</td>
<td>movw r1, :lower16:d</td>
</tr>
<tr>
<td>movt r1, :upper16:d</td>
<td>movt r1, :upper16:d</td>
</tr>
<tr>
<td>ldr r1, [r1]</td>
<td>ldr r1, [r1]</td>
</tr>
<tr>
<td>add r0, r0, r1</td>
<td>add r0, r0, r1</td>
</tr>
<tr>
<td>bx lr</td>
<td>bx lr</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

```
foo:
  mov  r2, lr
  bl   OUTLINED_FUNCTION_0
  mov  lr, r2
  add  r0, r0, r1
  bx   lr

bar:
  mov  r2, lr
  bl   OUTLINED_FUNCTION_0
  mov  lr, r2
  sub  r0, r0, r1
  bx   lr
```

```
OUTLINED_FUNCTION_0:
  movw r0, :lower16:a
  movt r0, :upper16:a
  ldr  r0, [r0]
  movw r1, :lower16:b
  movt r1, :upper16:b
  ldr  r1, [r1]
  add  r0, r0, r1
  movw r1, :lower16:c
  movt r1, :upper16:c
  ldr  r1, [r1]
  add  r0, r0, r1
  movw r1, :lower16:d
  movt r1, :upper16:d
  ldr  r1, [r1]
  bx   lr
```

28 bytes saved in this case
Non-Outlinable cases

- Conditional execution, IT blocks
- IP register or Condition code usage
- Instructions with PC dependent behavior
- Stack operations
Spec2K6 Cortex-A53 ARM mode results normalized to LLVM Oz

- Code size is 4.8% smaller on average with Machine Outliner turned on
Spec2K6 Cortex-A53 Thumb mode results normalized to LLVM Oz

- Code size is 3.3% smaller on average with Machine Outliner turned on for Thumb
LLVM CTMark Testsuite Cortex-A53 ARM mode results normalized to LLVM Oz

- Code size is 5.3% smaller on average with Machine Outliner turned on
LLVM CTMark Testsuite Cortex-A53 Thumb mode results normalized to LLVM Oz

- Code size is 2.8% smaller on average with Machine Outliner turned on
Zephyr Cortex-M4 results normalized to LLVM Oz

- Code size is 1.1% smaller on average with Machine Outliner turned on
- It brings LLVM to parity with GCC
Summary

- Clang is slightly worse than GCC for code-size on the Zephyr code-base.
- Clang can be substantially better for DSP/Floating point code.
- Forthcoming size optimizations to Clang such as VFE and the outliner will help to close the gap.
Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org
Backup, measuring code-size
Factors influencing the code size of a program

Compiler

```c
void f1(){
    libf1();
}
void f2(){
    libf2();
}
```

linker and libraries

```c
.text.f1
f1:
BL libf1
BX lr
.text.f2
f2:
BL libf2
BX lr
```

```c
libf1.o
.text
libf1:
BX lr
libf2.o
.text
libf2:
BX lr
```

Code generated by compiler.

Code from static libraries added at link time.

Linker optimizations such as garbage collection depend on section layout.
Measuring code size in an ELF file

```c
int global_variable = 10;
int func(void) {
    return global_variable;
}
```

- ELF sections can be read with `readelf --sections`
- ELF symbols can be read with `readelf --symbols`
- The size of a section includes all functions defined within
- Compiler gives symbol defining a size that can give the code size of a function

---

<table>
<thead>
<tr>
<th>Num</th>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Bind</th>
<th>Vis</th>
<th>Ndx</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>000000000</td>
<td>16</td>
<td>FUNC</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>3</td>
<td>func</td>
</tr>
<tr>
<td>5</td>
<td>000000000</td>
<td>4</td>
<td>OBJECT</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>7</td>
<td>global_variable</td>
</tr>
</tbody>
</table>
Compiler output is not enough, comdat groups

```cpp
// May be instantiated in other // files.
template<typename T>
T add(T a, T b) {
    return a+b;
}

// Take address of function // to force definition
template<typename T>
using fptr = T (*)(T, T);

fptr<int> fpint = &add<int>;

int func(int a, int b) {
    return add(1, 2);
}
```

```
.text
_Z4funcii:
ADD r0, r1, r0
BX lr

.group
SHT_GROUP
GRP_COMDAT
.text._Z3addIiET_S0_S0_%

.text._Z3addIiET_S0_S0_%_Z4funcii:
ADD r0, r1, r0
BX lr
```
Compiler output is not enough, library calls

```c
#include <string.h>
#define SIZE 12

int globvar1[SIZE];
int globvar2[SIZE];

void func() {
    memcpy(&globvar1, &globvar2, sizeof(globvar1));
}
```

// At SIZE 12 memcpy inlined
LDM ip!, {r0, r2, r3, r4, r5, lr}
STM r1!, {r0, r2, r3, r4, r5, lr}
LDM ip, {r0, r2, r3, r4, r5, lr}
STM r1, {r0, r2, r3, r4, r5, lr}

// At SIZE 120 memcpy called, smaller
// code size in object, but the
// executable will include size of
// __aeabi_memcpy4
BL 0 __aeabi_memcpy4
Cortex-M4 non-normalized results

- Flash size includes code and non ZI data
- Library size so small it is not visible
- Cortex-M4 has the most supported samples
- Total LLVM Size 3462456
- Total GCC Size 3424092
- LLVM is 1.1% larger
Cortex-M0 non-normalized results

- Cortex-M0 has fewer supported samples
- Total LLVM Size 2474544
- Total GCC Size 2404132
- LLVM is 2.8% larger
CMSIS DSP Cortex-M4f

CMSIS DSP Code Size

LLVM Code Size  GCC Code Size

Library

Controller  FastMath  Support  Statistics  Complex  Filtering  Matrix  Transform  Total