# Optimising *strlen*

## Setup

String length = 255

first look at the "hot path" (aka the loop)

00000008 <.bb0>:

...

b 44 <.bb1>

00000044 <.bb1>:

44: e5946000 ldr r6, [r4]

48: e59d0004 ldr r0, [sp, #4]

4c: e5908000 ldr r8, [r0]

50: e1a05006 mov r5, r6

54: e3006000 movw r6, #0

58: e3406000 movt r6, #0

5c: e5967000 ldr r7, [r6]

60: e0060798 mul r6, r8, r7

64: e0858006 add r8, r5, r6

68: e1a06008 mov r6, r8

6c: e5967000 ldr r7, [r6]

70: e3005000 movw r5, #0

74: e3405000 movt r5, #0

78: e5958000 ldr r8, [r5]

7c: e1570008 cmp r7, r8

80: e3a07000 mov r7, #0

84: 13007001 movwne r7, #1

88: e3570001 cmp r7, #1

8c: aa000000 bge 94 <.bb2>

90: ea000008 b b8 <.bb3>

00000094 <.bb2>:

94: e59d0004 ldr r0, [sp, #4]

98: e5907000 ldr r7, [r0]

9c: e3006000 movw r6, #0

a0: e3406000 movt r6, #0

a4: e5965000 ldr r5, [r6]

a8: e0878005 add r8, r7, r5

ac: e59d0004 ldr r0, [sp, #4]

b0: e5808000 str r8, [r0]

b4: eaffffe2 b 44 <.bb1>

000000b8 <.bb3>:

...

## Text Description automatically generated

This profiling data is not immediately super useful, at first glance it looks to be saying “loads are slow” which is sort of obvious.

## Peephole Optimisations

However what is interesting is that the multiply is essentially multiplying r6 \* 1 (which will resolve to just r6).

This is just one of many forms that could be simplified by the implementation of a peephole optimiser.

Adding in this simple transform (eliminating multiplications by one) provided significant performance increase:

-------------------------------------------------------------

Benchmark Time CPU Iterations

-------------------------------------------------------------

BM\_FUNCTION(Helix) 1366 ns 1366 ns 512201

BM\_FUNCTION(TCC) 1432 ns 1432 ns 488647

BM\_FUNCTION(Clang) 4.00 ns 4.00 ns 174892475

BM\_FUNCTION(GCC) 358 ns 358 ns 1957745

## Performance Deviations

Significant performance deviations when running individual compiler benchmarks vs running as a group:

Take these results

-------------------------------------------------------------

Benchmark Time CPU Iterations

-------------------------------------------------------------

BM\_FUNCTION(Helix) 1351 ns 1350 ns 518198

BM\_FUNCTION(TCC) 1432 ns 1432 ns 489087

BM\_FUNCTION(Clang) 4.00 ns 4.00 ns 174888501

BM\_FUNCTION(GCC) 358 ns 358 ns 1957711

Then compare to helix individual benchmark:

2022-02-16T17:53:22+00:00

Running ./strlen-helix

Run on (4 X 1500 MHz CPU s)

Load Average: 0.14, 0.18, 0.13

----------------------------------------------------------

Benchmark Time CPU Iterations

----------------------------------------------------------

SingleBenchmark 1058 ns 1058 ns 661424

Then for TCC individual

2022-02-16T17:56:50+00:00

Running ./strlen-tcc

Run on (4 X 1500 MHz CPU s)

Load Average: 0.12, 0.15, 0.12

----------------------------------------------------------

Benchmark Time CPU Iterations

----------------------------------------------------------

SingleBenchmark 954 ns 954 ns 734077

My guess is that since TCC & Helix depending on a lot of memory operations in the hot loop, whereas the others (clang/GCC) keep their values in registers for the whole run.

## Loading Immediate Values

First thing that I’m looking at is are the repeated movw/movt/ldr instruction seqeuences, which is what the compiler currently uses to load immediate values into register.

Note that the given assembly was disassembled using objdump, and that the movw/movt appears to be moving #0 (zero). This is because specific addresses haven’t been assigned to the “global” data it’s loading from.

Using GDB to look at the runtime assembly, shows that these do get the right values:

Text

Description automatically generated with medium confidence

Coming from x86 this series of instructions might seem a bit excessive just to load a 32 bit value into a register, however on ARM instructions are fixed width (32bits for ARMv7-A) and hence cannot fit a whole 32 bit immediate value into the instructions. On x86 instructions are not fixed width, and can accommodate almost arbitrary data.

This pattern works by storing the immediate value in “global” data, then loading the address of that global into a register (r6 in the examples above), and then using LDR to load the value from that address into the final register that contains the immediate.

One thing we can do here is eliminate the LDR by splitting the 32 bit constant in half and using movw/movt to load the immediate itself into the register instead of the address.

Firstly, add a new pass `arm-split-constants` that will perform this transform.

Given this simple test Case:  
int main() { return 45; }

It would (before the transform) generate the following assembly

Text

Description automatically generated

Then implementing a pass that scans for all constant integers in the IR and then instead of hoisting the constant to a global and loading from that, instead splits the constant using movw/movt now it looks something like this:

Text

Description automatically generated

Now to benchmark these changes

## Optimising compare/branch

A picture containing text

Description automatically generated

Can be optimised to:

Diagram

Description automatically generated

Didn’t provide any significant performance increase in the given strlen benchmark (I think this is due to perf being memory bound)

## Load/Store Optimisations

Now that the low hanging fruit are out the way & given all the above optimisations, profiling data now looks like this:

Text

Description automatically generated with medium confidence Text

Description automatically generatedText, application, chat or text message

Description automatically generated

Since perf does sample based programming, the percentages given above should be taken with a grain of salt, however what does seem obvious is that the main thing slowing us down now is the memory loads (& appropriate stores) in the hotpath.

These loads and stores, whilst might seem completely redundant to any programming whose programmed in assembly before, are a result of the simple way the IR is generated, C variables (that abstractly exist on the *stack*) we literally put on the stack. This simplifies code generation, however it is slow. Instead we can try and store variables in registers as much as possible, and only actually place them on the stack if we need to take their address.

Clang generates internally IR that looks a bit like we have here, e.g. emitting load/stores for every “stack” variable, however LLVM has a pass *Mem2Reg* which converts variables that don’t need to be in memory to existing as purely registers. Clang/LLVM also has it’s own reason for emitting this slow code from the frontend, and that is because LLVM IR is in SSA form and emitting code in SSA form would complicate the frontend significantly. Instead it is simpler to emit code that loads/stores to memory, and have the middle end optimise that out.

Our IR is not in SSA form, however it is still simpler to emit code that uses the stack for variables. It also means we don’t need an explicit addressof operation (stack\_alloc returns an address, which is what we use).

Time to implement our own Mem2Reg pass 😊

2022-02-17T14:20:27+00:00

Running build/strlen

Run on (4 X 1500 MHz CPU s)

Load Average: 0.10, 0.09, 0.09

-------------------------------------------------------------

Benchmark Time CPU Iterations

-------------------------------------------------------------

BM\_FUNCTION(Helix) 1355 ns 1352 ns 516915

BM\_FUNCTION(TCC) 1440 ns 1436 ns 487872

BM\_FUNCTION(Clang) 4.02 ns 4.01 ns 174870703

BM\_FUNCTION(GCC) 359 ns 358 ns 1957732

^ profiling looked like this before mem2reg (after the previous optimisations had been applied)

After mem2reg was added timings looked like this (and after register allocator fix & optimisations for the binary search algorithm)

[ 50%] Linking CXX executable strlen

[100%] Built target strlen

2022-02-20T17:45:20+00:00

Running build/strlen

Run on (4 X 1500 MHz CPU s)

Load Average: 0.30, 0.16, 0.10

-------------------------------------------------------------

Benchmark Time CPU Iterations

-------------------------------------------------------------

BM\_FUNCTION(Helix) 937 ns 937 ns 745278

BM\_FUNCTION(TCC) 1447 ns 1434 ns 488615

BM\_FUNCTION(Clang) 4.05 ns 4.01 ns 174875749

BM\_FUNCTION(GCC) 362 ns 358 ns 1957394

# Annex 1

## Original Timings (before any optimisation)

-------------------------------------------------------------

Benchmark Time CPU Iterations

-------------------------------------------------------------

BM\_FUNCTION(Helix) 1751 ns 1750 ns 399783

BM\_FUNCTION(TCC) 1435 ns 1434 ns 487950

BM\_FUNCTION(Clang) 4.00 ns 4.00 ns 174896358

BM\_FUNCTION(GCC) 358 ns 358 ns 1957812