Comparative Analysis of GCC Codegen for AArch64 and RISC-V

Paul-Antoine Arras

BayLibre, France

Abstract

This contribution explores possible improvements in GCC code generation for RISC-V. We collected dynamic instruction counts from selected SPEC CPU 2017 benchmarks and compared the results with AArch64. Findings reveal that prominent compiler weaknesses include missing instruction patterns, extra move instructions, and suboptimal register allocation. Additionally, addressing ISA limitations, such as the lack of a scaled addressing mode, and vectorising library functions, like `memset` *and mathematical operations, are crucial for maximising RISC-V efficiency.*

Experimental Setup

The study targeted ARM AArch64 SVE2 and RISC-V RV64GCV, and utilised the following tools:

- SPEC CPU 2017 v1.1.9;
- GNU toolchain (GCC) built from master on 8 August 2024;
- GNU C library (glibc) similarly built;
- User-mode QEMU v9.1.0-rc1.

The benchmarks were compiled for both architectures, executed with the reference data set, and analysed using QEMU with a custom plugin[1] designed to identify hotspots and measure dynamic instruction counts[2]. The analysis focused on the most frequently executed basic blocks, referred to as *top blocks*. The selected benchmarks exhibited the largest differences in dynamic instruction counts (DIC) between AArch64 and RISC-V: 502.gcc_r (41%), 510.parest_r (54%), 549.fotonik3d_r (39%), and 554.roms_r (36%).

Weaknesses and Deficiencies Identified

Compiler

Default LMUL-MAX In the RISC-V Vector (RVV) extension, the *l*ength *mul*tiplier (LMUL) determines how many registers can be operated on at once. LMUL can be set dynamically as long as it does not exceed LMUL_MAX, which defaults to 1 in GCC.

In some cases, the default LMUL_MAX restricts the parallelism level and thus increases the DIC. This is particularly stringent on the 549.fotonik3d_r benchmark: by default, RISC-V's DIC is 38.7% higher than AArch64's; but when LMUL_MAX is set to dynamic, RISC-V's DIC becomes almost 4% lower than AArch64's.

However, increasing LMUL_MAX might not be beneficial on real hardware, depending on the actual microarchitecture[3].

SIMD Clones for Mathematical Functions At the time of analysis, GCC does not produce any SIMD clones for RISC-V. On AArch64, the combination of vectorised implementations in glibc (see below) and SIMD cloning can be used. As a result, on the 554.roms benchmark, all calls to the `exp` mathematical function account for 28% fewer dynamic instructions on AArch64.

Vector-Scalar Instructions Some RVV instructions have a variant where one input operand is a scalar register. For example, the floating-point multiply-add instruction has the following definitions:

```
vfmadd.vv    vd, vs1, vs2, vm
vfmadd.vf    vd, rs1, vs2, vm
```

However, GCC tends to emit a vector-vector instruction plus a broadcast when a single vector-scalar instruction would have sufficed. For instance, we have:

```
vfmv.v.f     v6, fa4
vfmadd.vv    v4, v6, v21
```

Which could be simply:

```
vfmadd.vf    v4, fa4, v21
```

Extra Move Instructions Given a sequence of multiply-add operations, each reusing the previous result, a common idiom is to keep overwriting the same destination register to limit pressure and cut extra move instructions. We observed instances where GCC followed this idiom for AArch64 while, on RISC-V, it used many registers and added extra instructions:

[1] We thank Rivos for providing the plugin.

[2] Dynamic instruction count was chosen as a vendor-neutral metric, and the significant performance gap between the architectures reduces the risk of overfitting.

[3] https://gcc.gnu.org/PR114686

```
vmv1r.v      v4, v1
vfmadd.vv    v4, v3, v31
vmv1r.v      v5, v1
vfmadd.vv    v5, v4, v30
vmv1r.v      v10, v1
vfmadd.vv    v10, v5, v29
```

ISA

Missing Scaled Addressing Mode When iterating over an array, it is beneficial to be able to read at a specific index in one instruction given (1) the base address, (2) the index and (3) the element size. The effective address can then be computed as: `(1) + (2) « (3)`. AArch64 provides such an addressing mode, e.g.:

```
ld1w     {z30.s}, p7/z, [x3, x0, lsl #2]
```

On the other hand, RISC-V requires a separate instruction to perform the shift:

```
addi     t1, t1, 16
vle32.v  v4, (t1)
```

This is the most prevalent issue in floating-point benchmarks.

Scalar Register Loading AArch64 can load a pair of scalar registers from contiguous memory with one `ldp` instruction:

```
ldp    d24, d25, [x0, #16]
```

RISC-V has to use two consecutive instructions:

```
fld    fa1, 16(a5)
fld    fa5, 24(a5)
```

Bitfield Extract and Insert RISC-V's bit-manipulation standard extension (Zbs) only provides `bset` (single-bit set) and `bext` (single-bit extract). Extracting a bitfield requires two instructions: `slli` + `srli` (or `srai`). Inserting into a bitfield requires at least two instructions: `andi` + `slli`.

AArch64 has a richer set of bit manipulation instructions. For instance, general unsigned bitfield extraction can be achieved with the `ubfx` instruction, while `ubfiz` can be used for unsigned bitfield insert in zero.

Immediate Addressing Mode Several vector floating-point instructions lack an immediate addressing mode. For instance, `vfmax` compares the content of two vector registers and returns the maximum for each element. However, there is no way to directly compare a register with a constant immediate; an additional move instruction is required. The following example illustrates how to replace negative values with zeroes in a vector:

```
vmv.v.i      v3, 0
vfmax.vv     v2, v2, v3
```

In contrast, AArch64 can achieve the same result with a single `fmaxnm` instruction:

```
fmaxnm       z30.d, p7/m, z30.d, #0.0
```

Displacement Addressing Mode The encodings of vector load and store instructions do not have an immediate field. In other words, the only supported addressing mode is *register indirect*, rather than the more eclectic *displacement*. For example, to load a whole vector register from contiguous memory with a base address computed by adding an offset to the value of a scalar register, RISC-V requires two instructions:

```
addi          s1, s0, -496
vl1re64.v     v6, 0(s1)
```

While AArch64 can do it in a single instruction:

```
ldr           z27, [x29, #77, mul vl]
```

Library

Optimised `memset` glibc has hand-optimised implementations of heavily used functions for different subsets of the AArch64 ISA. However, at the time of analysis, glibc does not provide such optimised function implementations for RISC-V. For the 502.gcc benchmark, RISC-V spends at least 21% of the run in glibc's generic scalar `memset` implementation; AArch64 spends less than 1% of the run in its hand-optimised version of `memset`.

Optimised Mathematical Functions At the time of writing, glibc does not define any vectorised mathematical functions (libvecm) for RISC-V. The analysis of the 554.roms benchmark shows that AArch64 can leverage its Advanced SIMD implementation of the `exp` function 80% of the time. As a result, when dealing with double-precision floating-point numbers, a single call to this optimised variant can replace four similar calls to the original scalar version.

Future Work

We are currently implementing fixes for some of the weaknesses identified in the GCC RISC-V backend. In particular, we submitted a patch addressing the vector-scalar issue described above. Other areas of interest include loop unrolling and constant load hoisting.

Acknowledgements

This work has been carried out as a collaboration between BayLibre and Rivos Inc., and funded by the RISE Project.