Paul-Antoine
Arras


# RISC-V vs. AArch64

## Comparative Analysis of GCC Codegen


# Outline

- Context
- Experimental setup
- Results overview
- Weaknesses and deficiencies
  - Compiler
  - ISA
  - Library
- Implementation
- Future work

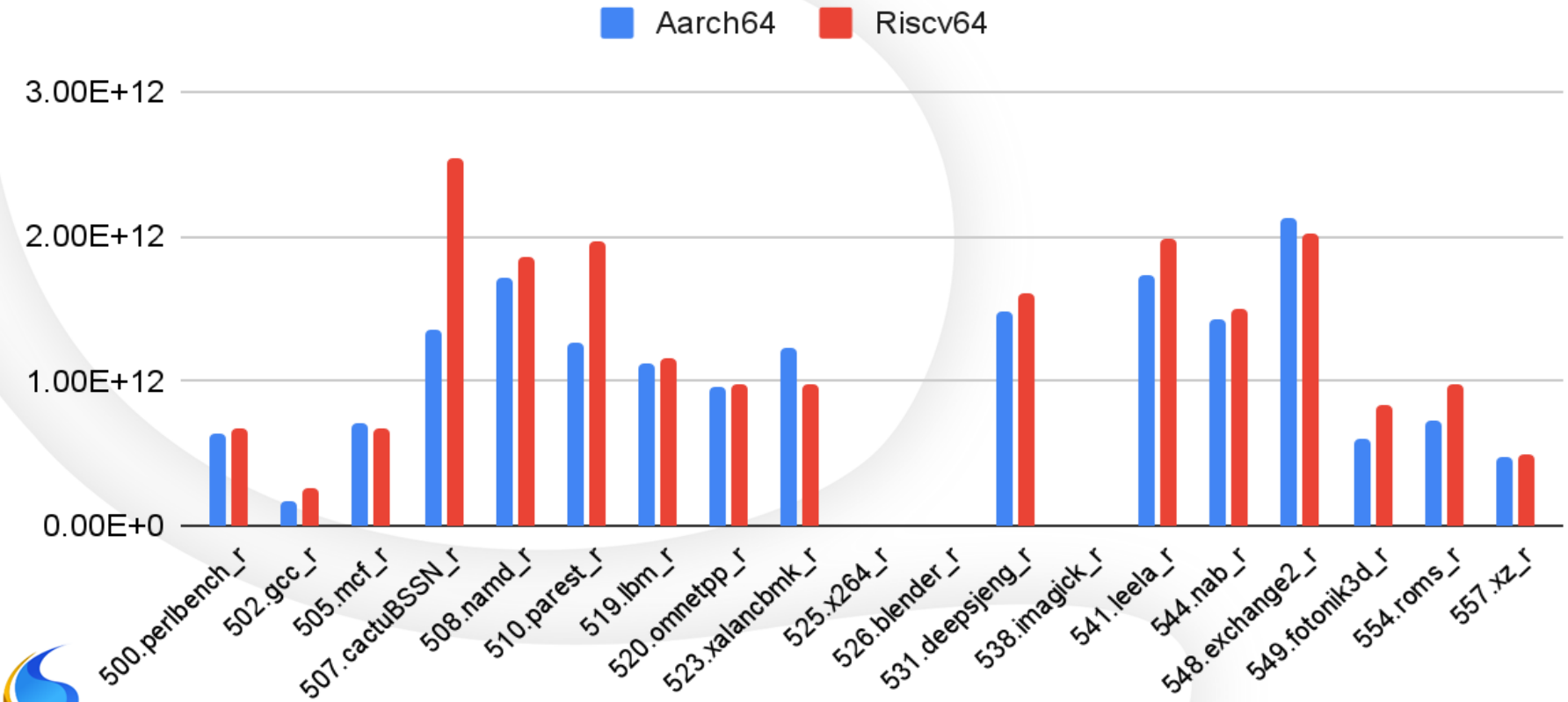
# Context

- Collaboration: BayLibre + Rivos + RISE
- Fact: RISC-V GCC trails AArch64 GCC
- SPEC benchmarks
- How to improve compiler codegen?
- Large gains can be made
- Vendor-neutral metric: *dynamic instruction count* (DIC)
- Analyse *assembly* in hotspots (aka top blocks)

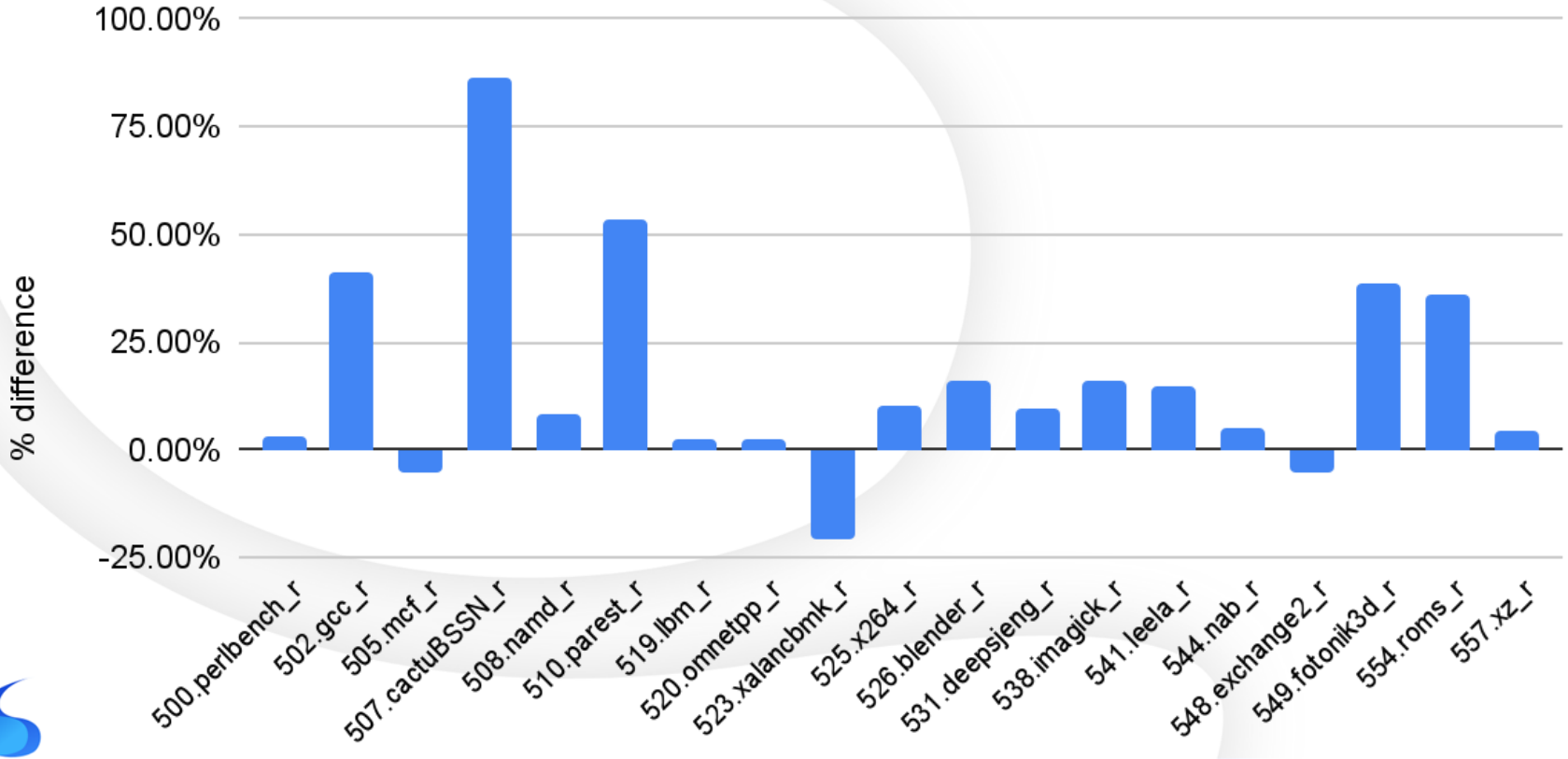
# Experimental setup

- Target architectures
  - ARM64 (aka AArch64) SVE2
  - RISC-V RV64GCV
  - 256-bit vectors
- SPEC CPU 2017 v1.1.9, *ref* data set
- GNU toolchain (8 Aug 24)
  - AArch64: `-mabi=lp64 -march=armv9-a+sve2 -msve-vector-bits=256`
  - RISC-V: `-mabi=lp64d -march=rv64gcv_zvl256b_zba_zbb_zbs_zicond -mrvv-vector-bits=zvl`
- User-mode QEMU v9.1.0-rc1 with *Rivos* plugin
  - AArch64: `QEMU_CPU = max,sve256=on`
  - RISC-V: `QEMU_CPU = rv64,vlen=256,v=true,vext_spec=v1.0,zve32f=true,zve64f=true,zba=true,zbb=true,zbc=true,zbs=true,zicond=true,zfhmin=true`

# Dynamic instruction counts



# RISC-V DIC vs. AArch64




SPEC CPU 2017 benchmark

# ISA: Missing scaled addressing mode

- 510.parest_r
  - DIC **+54%** vs. AArch64
  - Top block #0
    - **56%** of the run

```
while (val_ptr != val_end_of_row)
  s += *val_ptr++ * src[*colnum_ptr++];
```

- 3x dereference-increment
- Three operands
  - (1) Base address
  - (2) Index
  - (3) Element size: 4 bytes
- Effective address: (1) + (2) << (3)

**AArch64**

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

**RISC-V**

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

# Compiler: Vector-scalar instructions

- `vfmadd` has two flavours
  - Vector-vector: `vfmadd.vv vd, vs1, vs2, vm`
  - Vector-scalar: `vfmadd.vf vd, rs1, vs2, vm`
- 554.roms_r

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

- Combine into

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

- Patch submitted

# Compiler: Vector-scalar instructions

- Patch submitted
  - `define_insn_and_split`
  - Cost model
- DIC reduction
  - 554.roms_r
    - BB: -7.6%
    - Total: -0.27%
  - 503.bwaves_r: **-4.7%**
- Time on BananaPi-F3
  - Average over 3 runs
  - 503.bwaves_r: -0.33%

# Compiler + Library: memset

- 502.gcc_r: glibc's memset
  - RISC-V: 21% (7% with recent patches)
  - AArch64: <1%
- Strategy for builtins
  - *Expand* if possible (AArch64: 5 insns)
  - Glibc → last resort
    - Hand optimised
    - Generic scalar → last resort (RISC-V: 46 insns)
- What prevents the compiler from expanding the call?
  - LMUL = 1
  - Extensions Zicboz and Zic64b not available
  - `-finline-stringops` is not set

# ABI + Compiler + Library: SIMD math functions

- 554.roms_r: `__exp`
  - AArch64: 1.55% of the run
  - RISC-V: 6.48%
- ↑ libm's generic scalar
- Target-specific vectorised exp (libmvec)
  - AArch64: 80% of calls to `__builtin_exp` → `_ZGVnN2v_exp`
  - RISC-V: none → **32% more instructions**
- How RISC-V can catch up
  - RISC-V ELF psABI: name mangling
  - GCC: pass *simdclone* enabled by target hook `compute_vecsize_and_simdlen`
  - glibc: vector implementation

# Ongoing and future work

- Vector-scalar instructions
  - Patch ready
  - Push to trunk when stage 1 reopens
- Loop unrolling
  - Compare to AArch64
  - Opportunities might be missed
- Reorder vfmadd operands to avoid extra moves
- Hoist constant loading to loop preamble

# Thank you

Questions?

pa@baylibre.com