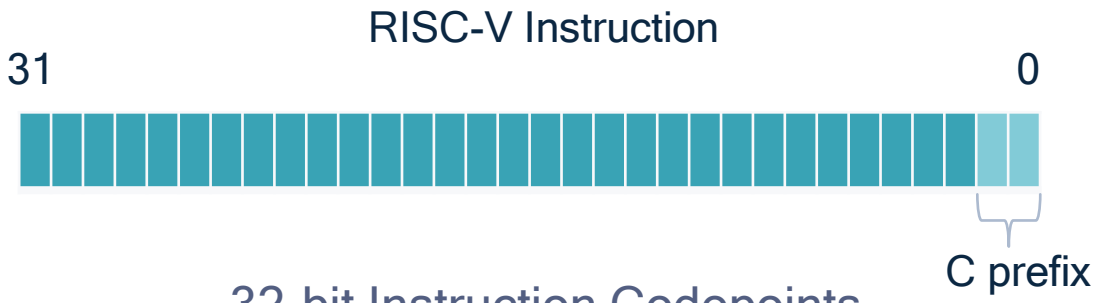
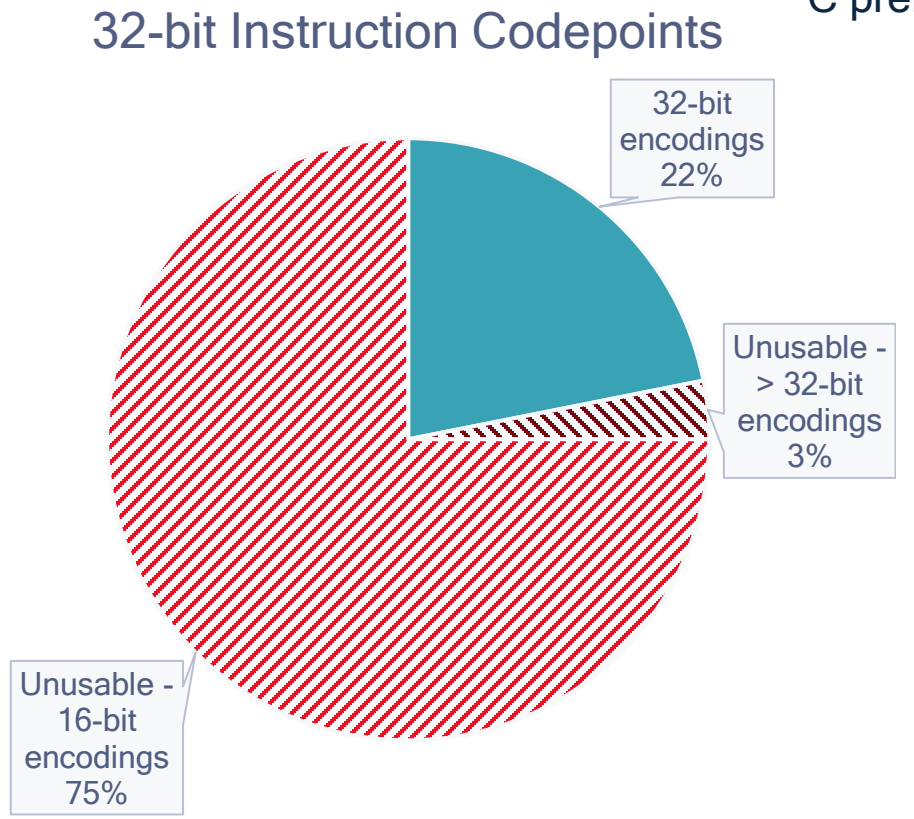
Qualcomm

A Case to Remove C from App Profiles

Derek Hower, James Ball, Conrado Blasco, Manu Gulati
Qualcomm Technologies, Inc.

C Extension – Compressed (16-bit) instruction encodings





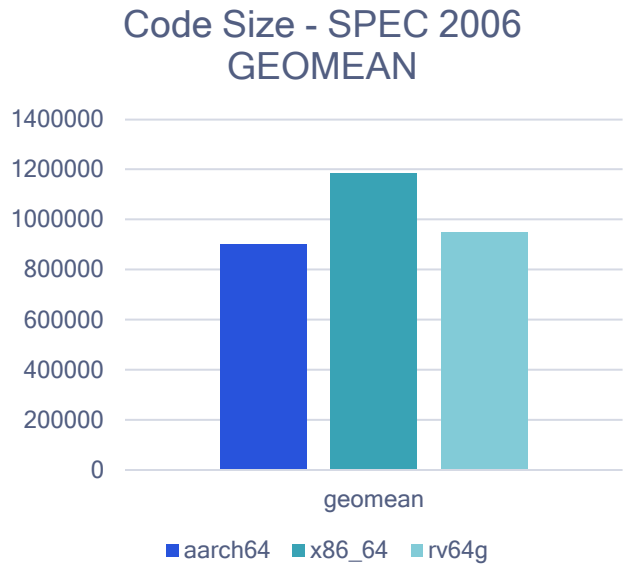
Facts

- C defines compressed re-encodings of common 32-bit instructions
  - **No new instructions** – always 1:1 mapping between 16/32-bit
- 2 LSBs of instruction identify length
  - 3 of 4 values indicate 16-bit
  - 1 of 4 values indicate >= 32-bit
- C binaries are packed: **32-bit encodings can be unaligned**

Problems with C in high performance designs

Pushes RISC-V to large opcodes

- RISC-V is nearly out of 32-bit opcodes
- Move to > 32-bit opcodes will *degrade code size*

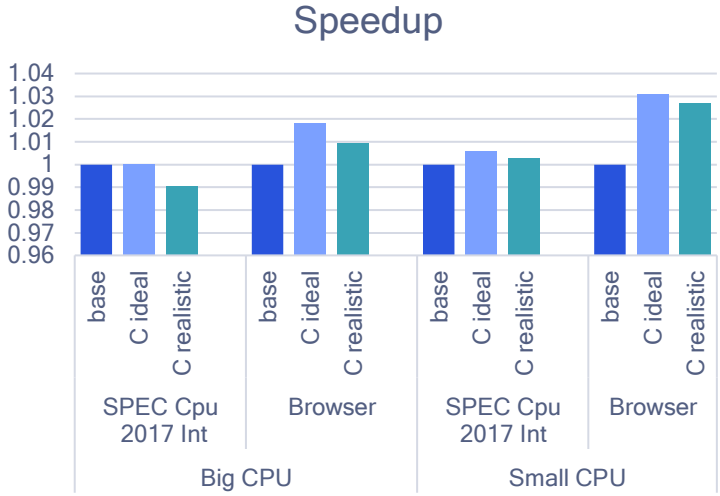


Substantial design complexity

- Unaligned fetch is challenging to design, verify
  - Cache line, page crossing instructions
  - Increased wire delay/muxing
- Leads to designs that are:
  - more expensive (NRE)
  - slower (extra pipe stages)
  - buggy (see Intel Jump Code Conditional)

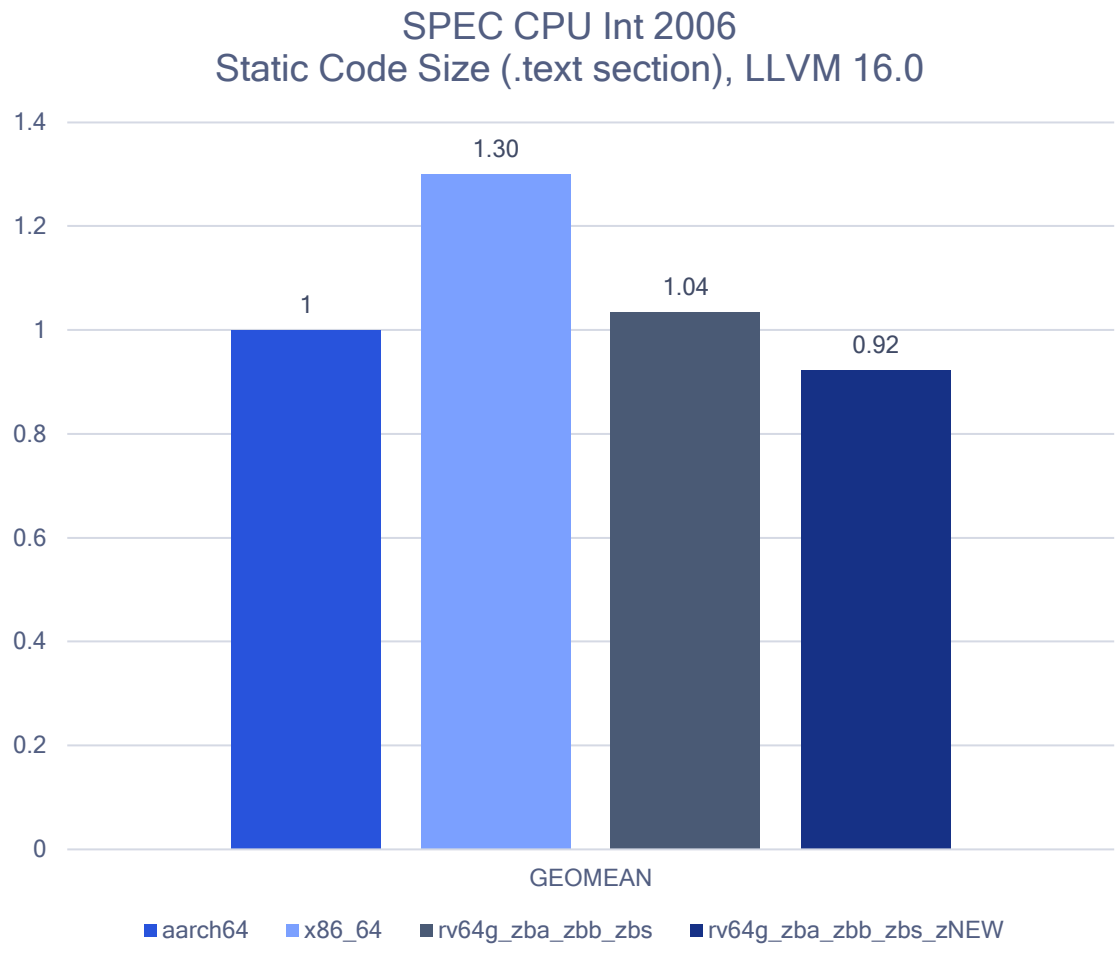
Performance benefit is modest

- Best case: 2-3% speedup
- Often: slowdown (net negative when program fits in icache)

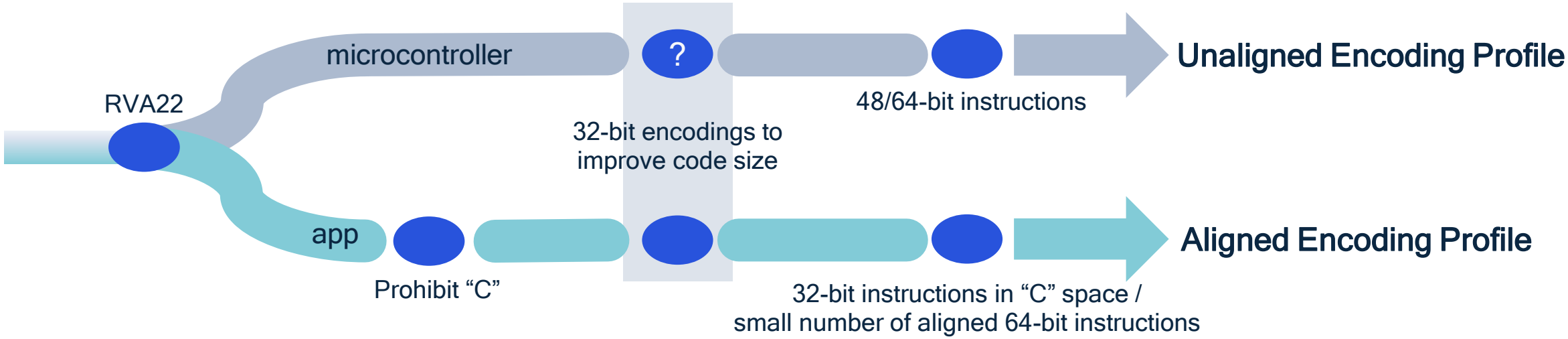


C is not needed

- RV64G is already competitive on code size
- RV64G + 32-bit instructions for code size is *best in class*
  - More ld/st addressing modes
  - Ld/st pair
  - Conditional immediate branches
  - Move pair
- RV64GC has little performance benefit in beefy designs



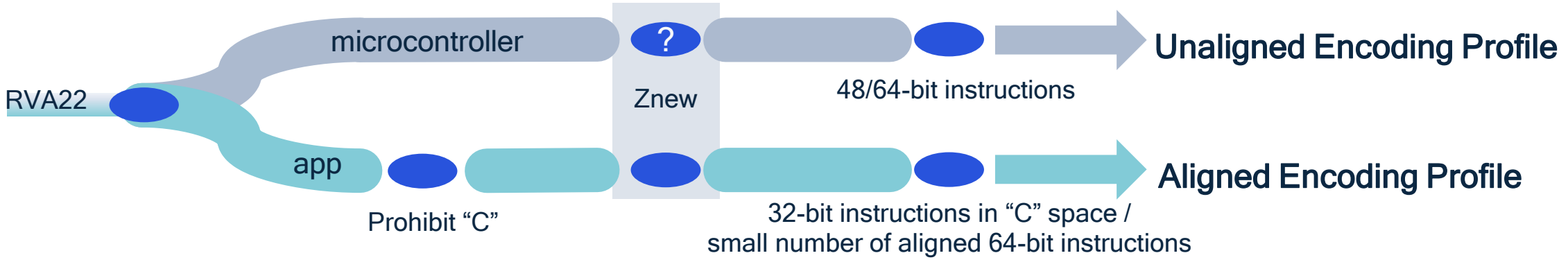
Diverging profiles



- Remove C from application profiles
- Once removed, the C opcode space can be reclaimed to keep code size down long term
- **It’s not too late**: commercial distros have not picked a base
  - But, time is running short

Meeting Break, resuming 10/5/2023

C in RISC-V Profiles, Continued

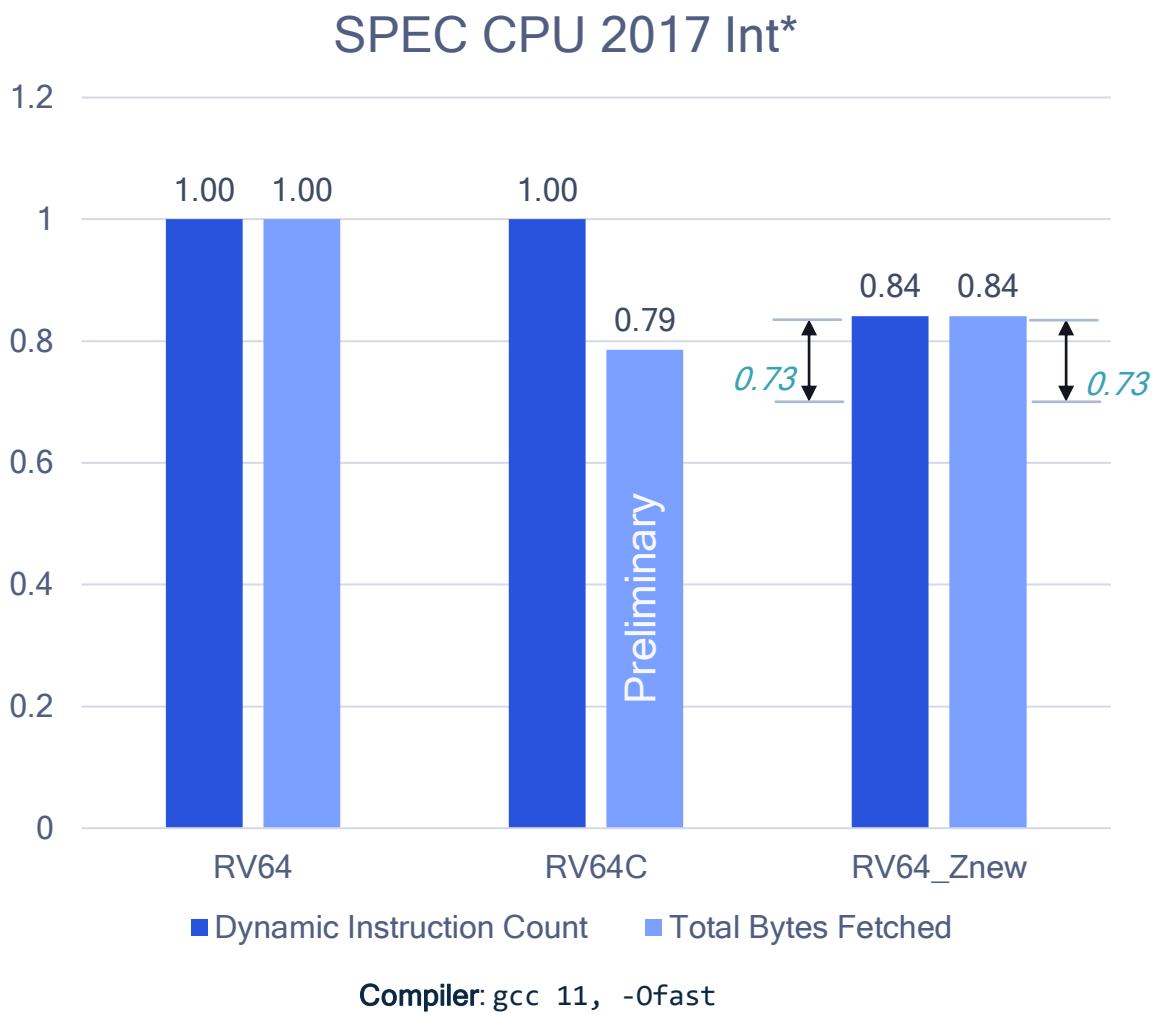


- Recap:
  - C extension consumes 75% of 32-bit instruction codepoints
  - C has small (in some cases negative) upside in apps processors
  - C is difficult to design/verify, will lead to bugs
  - RISC-V should split profile lineages
- Now:
  - Evaluate RISC-V dynamic instructions:
    - With C
    - With Znew
    - Compared to AArch64

Znew Instructions

- Load/store addressing modes
  - Register-register
  - Scaled register-register
  - PC-relative
- Pre/post increment load/stores
- Load/store pair
- Move pair
- Conditional immediate branch

RISC-V Dynamic Instruction Evaluation

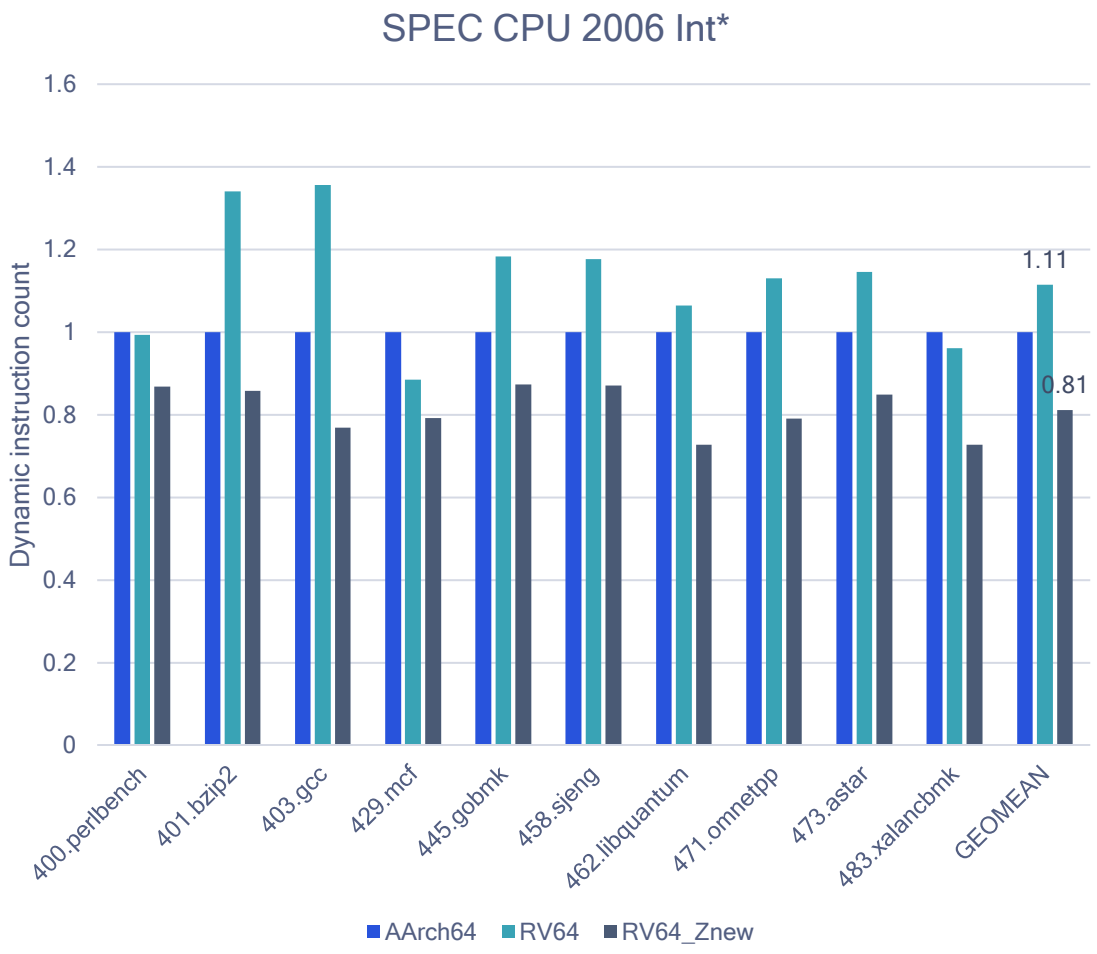


Compiler: `gcc 11, -Ofast`

* Excludes x264 (Compiler does not vectorize)

- Znew evaluation methodology:
  - Analyze RISC-V instruction stream, find/replace sequences that have single Znew equivalent
    - Conservative: no compiler help generating sequences
    - *Forward-looking*: with compiler help (next slide)
- Metrics
  - Dynamic instruction count → pipeline pressure
  - Total bytes fetched → icache pressure
- Result:
  - C extension does not affect dynamic instruction count (no new instructions)
  - Znew is effective at reducing dynamic instructions
  - C extension reduces the number of bytes fetched
    - Using **75%** of 32-bit encoding space
  - **Znew reduces dynamic instruction pressure in both metrics**
    - Using **< 1%** of 32-bit encoding space

Dynamic Comparison to AArch64



Compiler: `LLVM/Clang 12.0, -O3`

* 456.hmmer and 464.h264ref removed to avoid vector

- Forward-looking Methodology:
  - Analyze AArch64 instruction stream, pseudo-translate each instruction into RV equivalent(s)
    - Neutralizes compiler differences (even within same toolchain)
    - Evaluation with compiler that knows about many Znew instructions
  - Evaluation methodology validated to < 5% average error
- Result:
  - RV64, *with or without C*, has more dynamic instructions
    - But not always: mcf is branch-heavy, benefits from RV fused compare/branch
  - RV64_Znew has ~19% fewer dynamic instructions
    - Better than AArch64 because most instructions have 1:1 mapping, plus RV has fused compare/branch

Thank you

Qualcomm

Follow us on:

For more information, visit us at:

qualcomm.com & qualcomm.com/blog

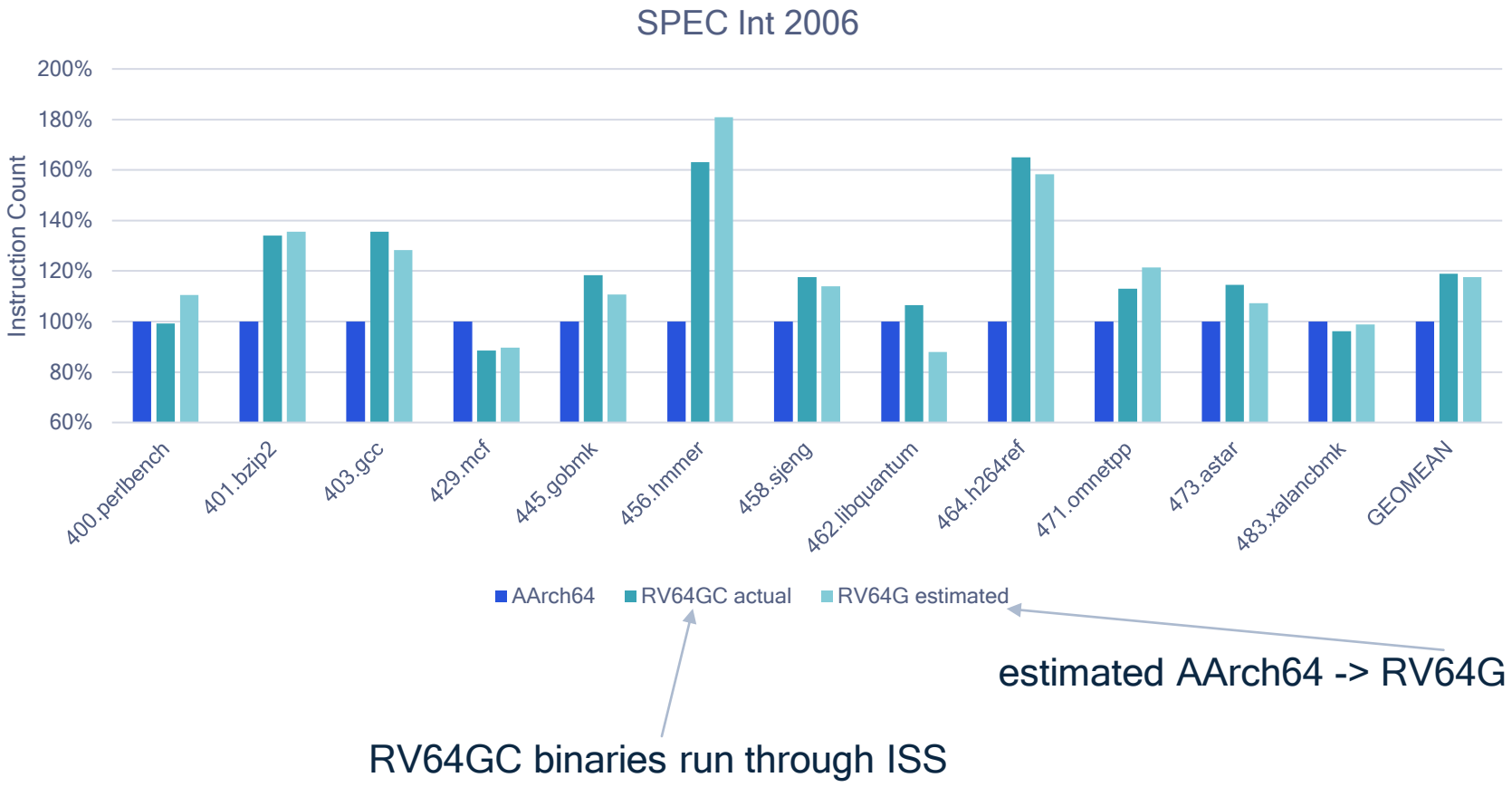
Znew Forward-looking Evaluation Strategy

- ~~Ideal: implement in compiler, compare to AArch64~~ *too much effort at this point*
- Strategy: pretend we can binary translate optimized AArch64 to RISC-V, evaluate code size
- Baseline:
  - AArch64 load or store expands to one or more RISC-V instructions
  - AArch64 cmp/br pairs collapse to one (or more, target is ≥ 4KiB away) RISC-V instructions
  - AArch64 vector instructions expand to num_elements RISC-V instructions
  - AArch64 DC ZVA expands to 64/8 = 8 RISC-V instructions
  - All other AArch64 instructions translate to exactly one RISC-V instruction
- Znew:
  - Same, but account for Znew in translation

- Validate:
  - Estimated RISC-V code size w/o Znew should match actual RISC-V code size

Validation

Methodology is valid if green bars are similar height



RV64GC binaries run through ISS

estimated AArch64 -> RV64G translation (no Znew)

- Validation: RV64 mean actual/estimated within tolerance
- RV64GC actual is sometimes better in branch-heavy code