RISC-V SUMMIT
NORTH AMERICA 2025

Unlocking 15% More Performance: A Case Study in LLVM Optimization for RISC-V

Phd Mikhail R. Gadelha
Igalia

RISC-V SUMMIT
NORTH AMERICA 2025

Unlocking ~~15%~~ More Performance: A Case Study in LLVM Optimization for RISC-V

Phd Mikhail R. Gadelha
Igalia

RISC-V SUMMIT

NORTH AMERICA 2025

Unlocking ~~15%~~ 16% More Performance: A Case Study in LLVM Optimization for RISC-V

Phd Mikhail R. Gadelha

Igalia

Up to 16% faster SPEC CPU® 2017 on SpacemiT-X60

RISC-V SUMMIT NORTH AMERICA 2025

- This RISE project was a focused, ten-month effort to optimize the LLVM compiler for RISC-V.

Up to 16% faster SPEC CPU® 2017 on SpacemiT-X60

- This RISE project was a focused, ten-month effort to optimize the LLVM compiler for RISC-V.
- Our target board was the Banana Pi BPI-F3 with a SpacemiT-X60 8-core RISC-V processor:
  - In-order processor.
  - Supports the RVA22U64 Profile and 256-bit RVV 1.0 standard.

Up to 16% faster SPEC CPU® 2017 on SpacemiT-X60

- This RISE project was a focused, ten-month effort to optimize the LLVM compiler for RISC-V.
- Our target board was the Banana Pi BPI-F3 with a SpacemiT-X60 8-core RISC-V processor:
  - In-order processor.
  - Supports the RVA22U64 Profile and 256-bit RVV 1.0 standard.
- Our goal: to close the performance gap between LLVM and the GCC compiler.

Up to 16% faster SPEC CPU® 2017 on SpacemiT-X60

- This RISE project was a focused, ten-month effort to optimize the LLVM compiler for RISC-V.
- Our target board was the Banana Pi BPI-F3 with a SpacemiT-X60 8-core RISC-V processor:
  - In-order processor.
  - Supports the RVA22U64 Profile and 256-bit RVV 1.0 standard.
- Our goal: to close the performance gap between LLVM and the GCC compiler.
- Our result: individual contributions boosted performance by up to 16% on SPEC CPU® 2017 benchmarks.

The Project

- Prior to this work, a clear performance gap existed between code generated by LLVM and GCC for RISC-V.

The Project

- Prior to this work, a clear performance gap existed between code generated by LLVM and GCC for RISC-V.
- There is no single solution to close the gap, as improvements and regressions occur daily within the codebase.

The Project

- Prior to this work, a clear performance gap existed between code generated by LLVM and GCC for RISC-V.
- There is no single solution to close the gap, as improvements and regressions occur daily within the codebase.
- This presentation will focus on our three main contributions to help close the gap:
  - Introducing a scheduling model for the SpacemiT-X60.
  - Improvements to vectorization across calls.
  - Register Allocation with IPRA Support for RISC-V.

Our Contributions

(major contributions first)

SpacemiT-X60 Scheduling Model

SpacemiT-X60 Scheduling Model

- Instruction Scheduling == Performance.
- Wrong latencies/resources → compiler makes poor choices.

```
fld     ft0, 0(a0)
fadd.d  ft1, ft0, ft2
fmul.d  ft3, ft4, ft5
```

before

```
fld     ft0, 0(a0)
fmul.d  ft3, ft4, ft5
fadd.d  ft1, ft0, ft2
```

after

SpacemiT-X60 Scheduling Model

- Instruction Scheduling == Performance.
- Wrong latencies/resources → compiler makes poor choices.

Load latency not modelled

```
fld     ft0, 0(a0)
fadd.d  ft1, ft0, ft2
fmul.d  ft3, ft4, ft5
```

before

```
fld     ft0, 0(a0)
fmul.d  ft3, ft4, ft5
fadd.d  ft1, ft0, ft2
```

after

SpacemiT-X60 Scheduling Model

- Instruction Scheduling == Performance.
- Wrong latencies/resources → compiler makes poor choices.

Dependent on ft0 → stalls

```
fld     ft0, 0(a0)
fadd.d  ft1, ft0, ft2
fmul.d  ft3, ft4, ft5
```

before

```
fld     ft0, 0(a0)
fmul.d  ft3, ft4, ft5
fadd.d  ft1, ft0, ft2
```

after

SpacemiT-X60 Scheduling Model

- Instruction Scheduling == Performance.
- Wrong latencies/resources → compiler makes poor choices.

```
fld     ft0, 0(a0)
fadd.d  ft1, ft0, ft2
fmul.d  ft3, ft4, ft5
```

Issued later, independent

before

```
fld     ft0, 0(a0)
fmul.d  ft3, ft4, ft5
fadd.d  ft1, ft0, ft2
```

after

SpacemiT-X60 Scheduling Model

- Instruction Scheduling == Performance.
- Wrong latencies/resources → compiler makes poor choices.

```
fld     ft0, 0(a0)
fadd.d  ft1, ft0, ft2
fmul.d  ft3, ft4, ft5
```

before

Load issued

```
fld     ft0, 0(a0)
fmul.d  ft3, ft4, ft5
fadd.d  ft1, ft0, ft2
```

after

SpacemiT-X60 Scheduling Model

- Instruction Scheduling == Performance.
- Wrong latencies/resources → compiler makes poor choices.

```
fld     ft0, 0(a0)
fadd.d  ft1, ft0, ft2
fmul.d  ft3, ft4, ft5
```

before

Independent op fills load latency

```
fld     ft0, 0(a0)
fmul.d  ft3, ft4, ft5
fadd.d  ft1, ft0, ft2
```

after

SpacemiT-X60 Scheduling Model

- Instruction Scheduling == Performance.
- Wrong latencies/resources → compiler makes poor choices.

```
fld     ft0, 0(a0)
fadd.d  ft1, ft0, ft2
fmul.d  ft3, ft4, ft5
```

before

Now executes with result ready

```
fld     ft0, 0(a0)
fmul.d  ft3, ft4, ft5
fadd.d  ft1, ft0, ft2
```

after

How We Got the Numbers

- We built custom microbenchmarks to measure instruction latencies.

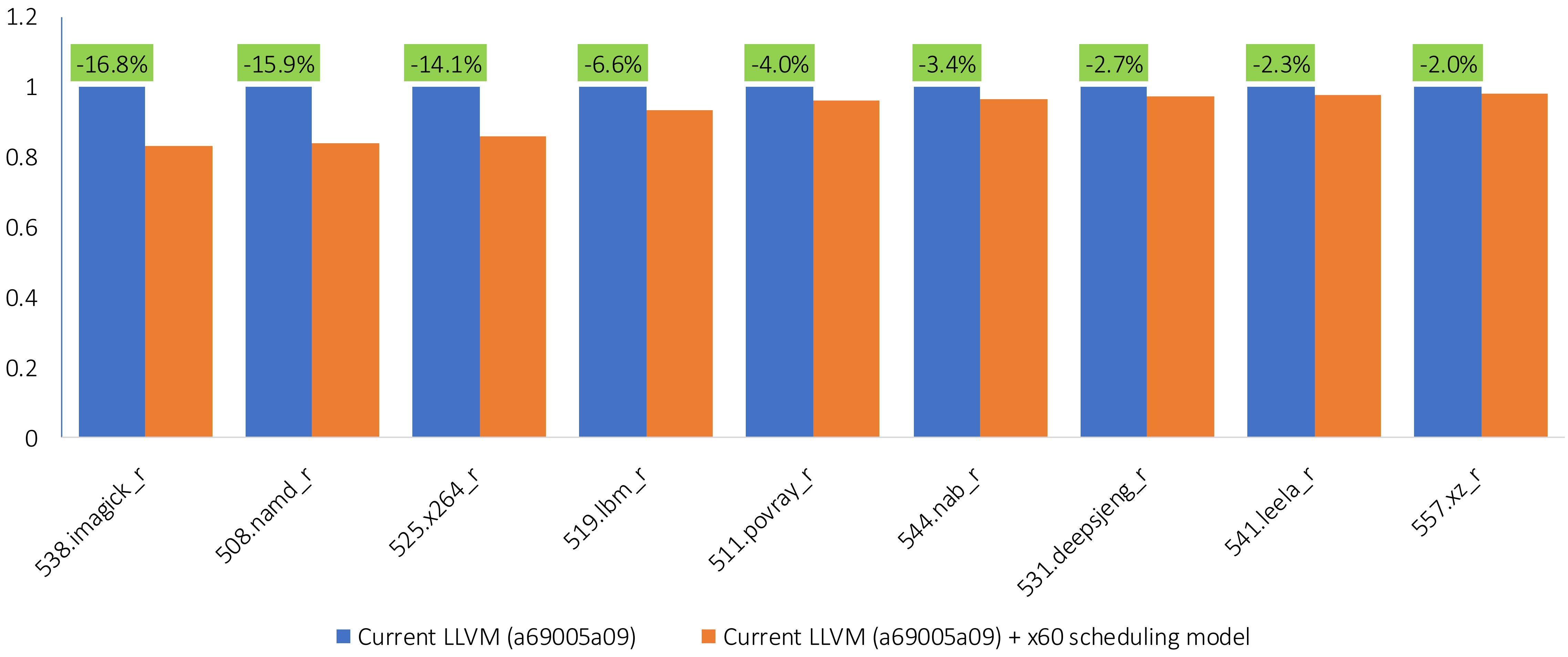
How We Got the Numbers

- We built custom microbenchmarks to measure instruction latencies.
- Most of instruction throughput data available at https://camel-cdr.github.io/rvv-bench-results/bpi_f3/index.html.

How We Got the Numbers

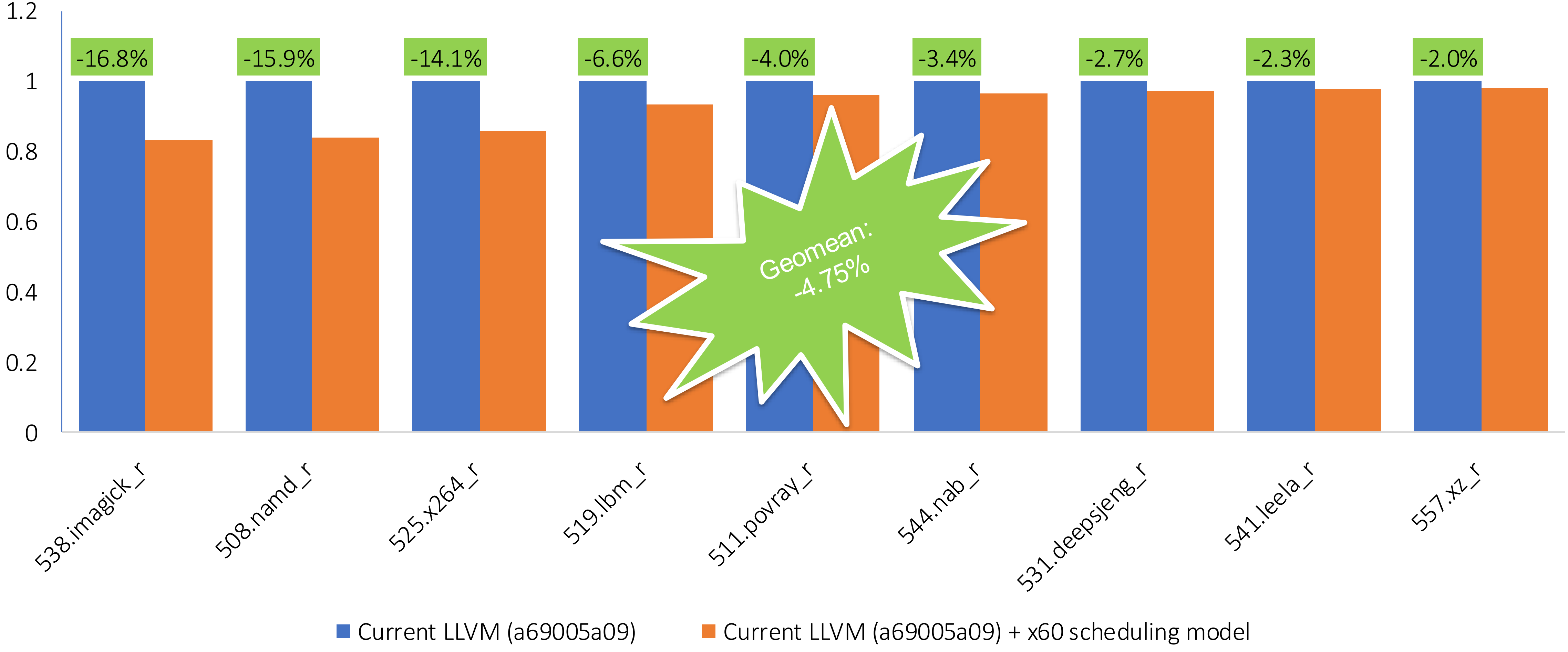
- We built custom microbenchmarks to measure instruction latencies.
- Most of instruction throughput data available at https://camel-cdr.github.io/rvv-bench-results/bpi_f3/index.html.
- It’s RISC but:
  - 201 scalar instructions.
  - 82 floating-point instructions.
  - 9185 RVV instructions (because of the combination of different LMULs and SEWs).

RVA22U64 SPEC exec time, O3+LTO+mcpu=spacemit-x60

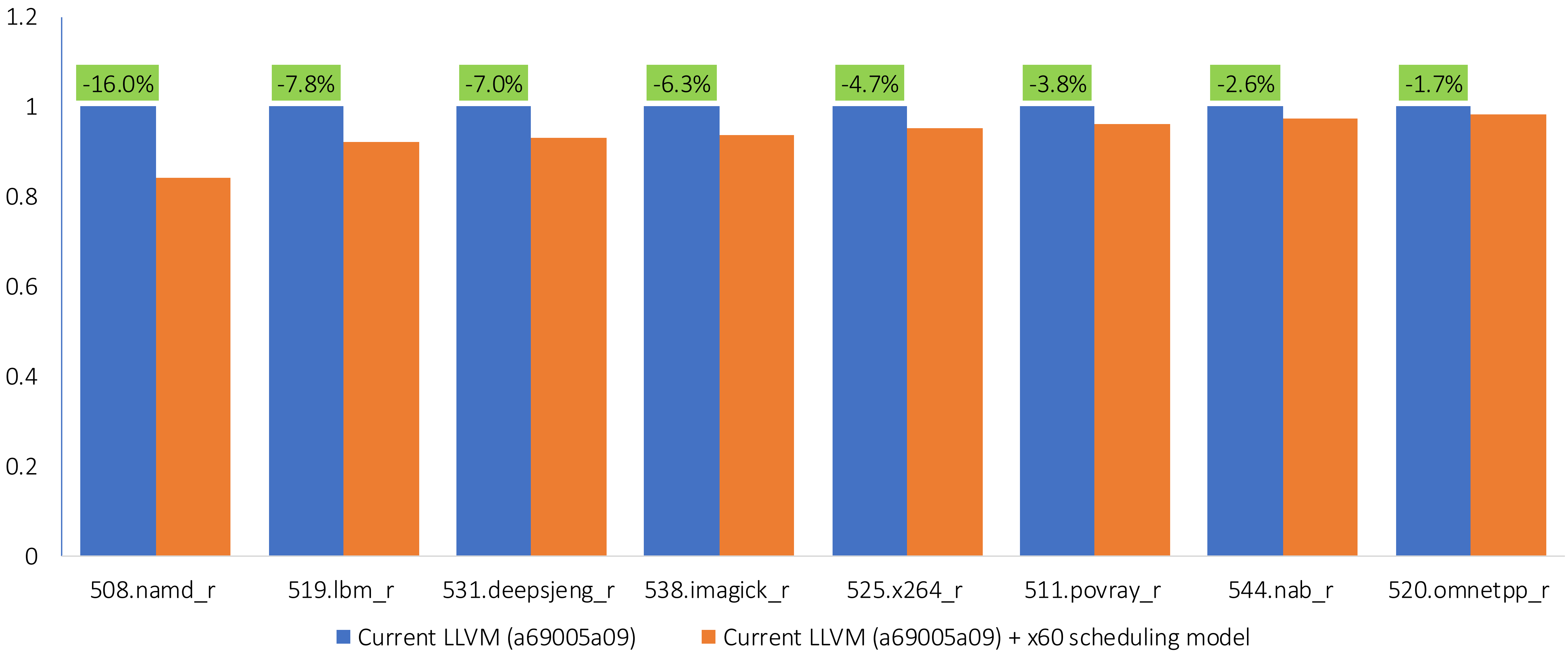


RVA22U64 SPEC exec time, O3+LTO+mcpu=spacemit-x60

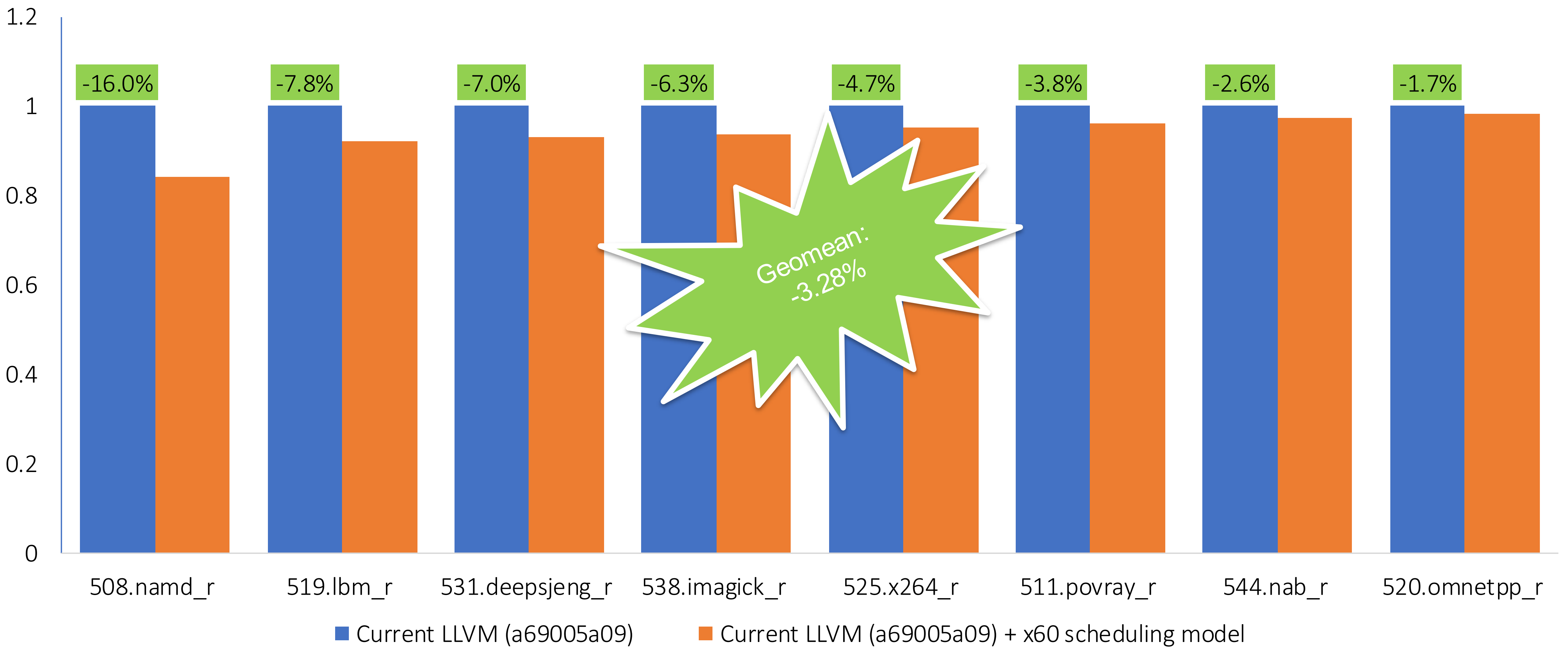
RISC-V SUMMIT NORTH AMERICA 2025



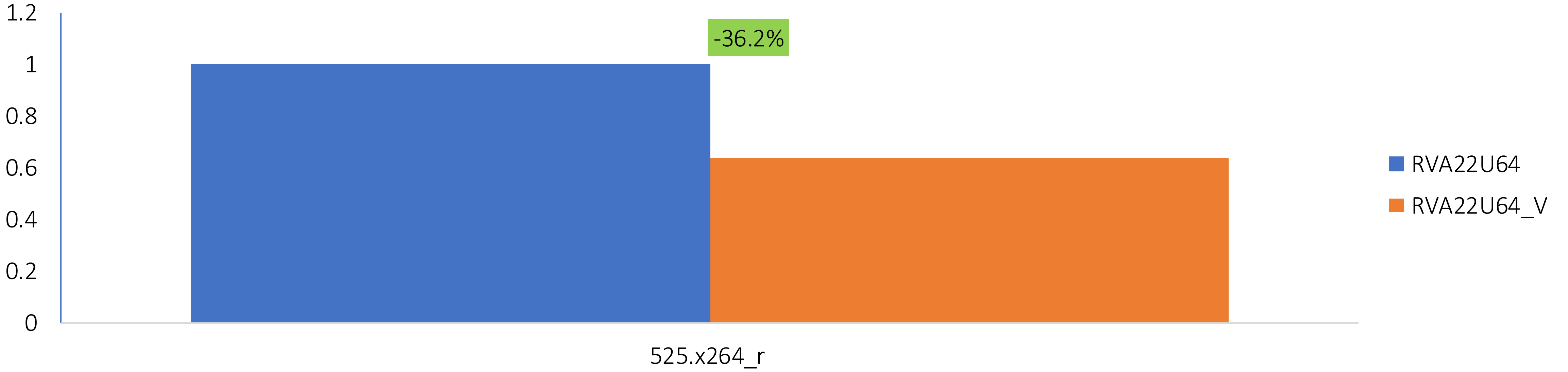
RVA22U64_V SPEC exec time, O3+LTO+mcpu=spacemit-x60



RVA22U64_V SPEC exec time, O3+LTO+mcpu=spacemit-x60

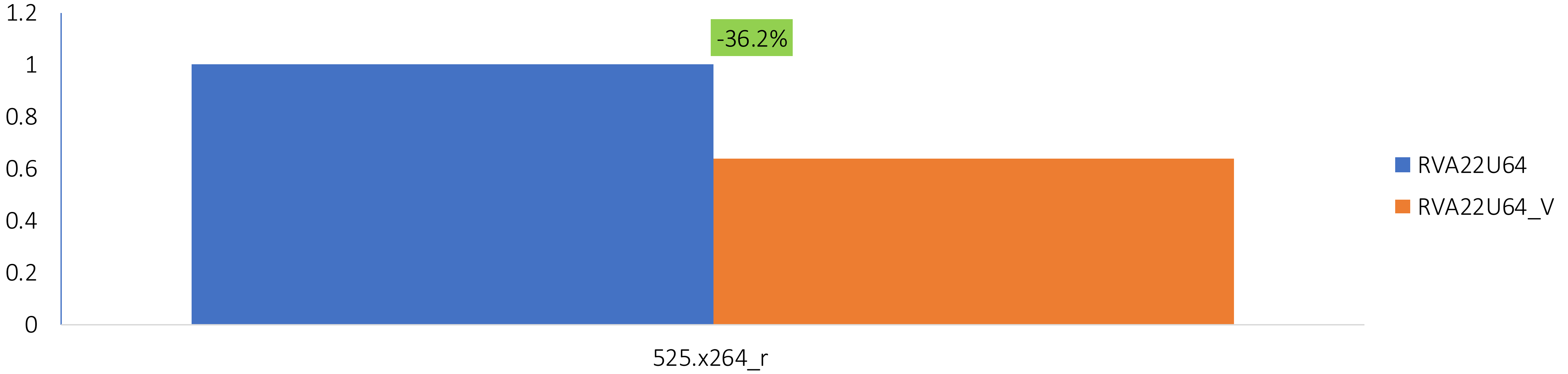


RVA22U64 vs RVA22U64_V, O3+LTO+mcpu=spacemit-x60



- Scheduling nearly eliminated the gap between scalar and vector configs.

RVA22U64 vs RVA22U64_V, O3+LTO+mcpu=spacemit-x60



- Scheduling nearly eliminated the gap between scalar and vector configs.
- On in-order processors like X60, scheduling is critical; on out-of-order, impact would be smaller and vectorization more decisive.

Improvements to Vectorization Across Calls

Improvements to Vectorization Across Calls

RISC-V SUMMIT NORTH AMERICA 2025

- Initial surprise: vectorized code sometimes underperformed scalar.

Improvements to Vectorization Across Calls

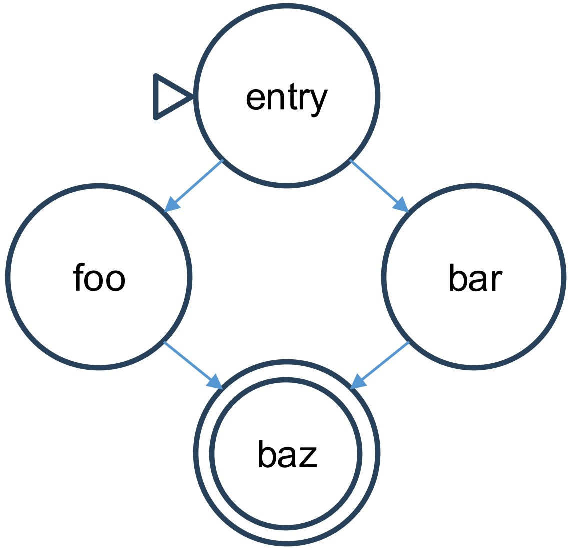
- Initial surprise: vectorized code sometimes underperformed scalar.
- Root cause: poor cost modeling and suboptimal spill behavior.

Improvements to Vectorization Across Calls

- Initial surprise: vectorized code sometimes underperformed scalar.
- Root cause: poor cost modeling and suboptimal spill behavior.
- The extra cycles were due to register spilling, particularly around function call boundaries.

The 544.nab_r case

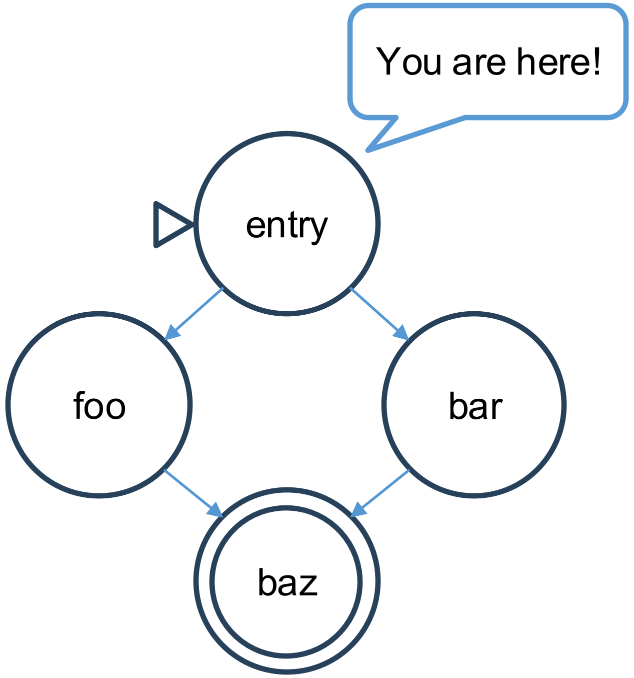
```
define void @f(i1 %c, ptr %p, ptr %q) {
entry:
  %x0 = load i64, ptr %p
  %p1 = getelementptr i64, ptr %p, i64 1
  %x1 = load i64, ptr %p1
  br i1 %c, label %foo, label %bar
foo:
  call void @g()
  br label %baz
bar:
  call void @g()
  br label %baz
baz:
  store i64 %x0, ptr %q
  %q1 = getelementptr i64, ptr %q, i64 1
  store i64 %x1, ptr %q1
  ret void
}
```



The 544.nab_r case

Loads first value from %p

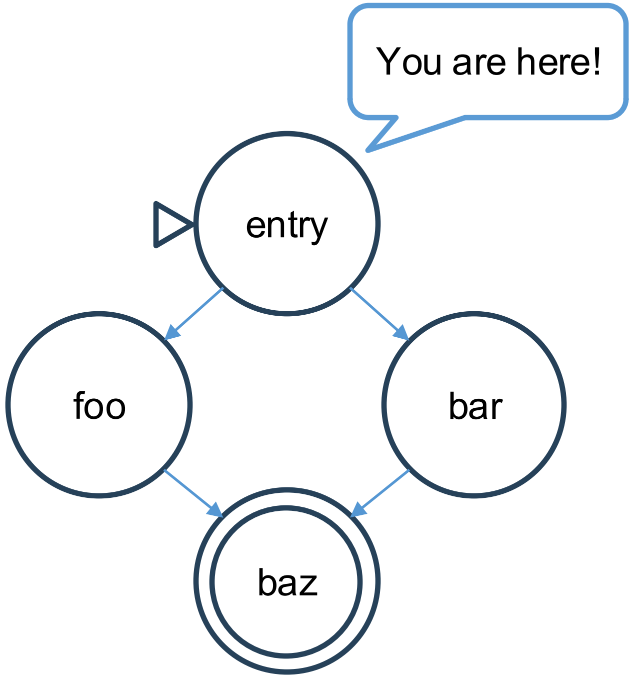
```
define void @f(i1 %c, ptr %p, ptr %q) {
entry:
  %x0 = load i64, ptr %p
  %p1 = getelementptr i64, ptr %p, i64 1
  %x1 = load i64, ptr %p1
  br i1 %c, label %foo, label %bar
foo:
  call void @g()
  br label %baz
bar:
  call void @g()
  br label %baz
baz:
  store i64 %x0, ptr %q
  %q1 = getelementptr i64, ptr %q, i64 1
  store i64 %x1, ptr %q1
  ret void
}
```



The 544.nab_r case

Loads second value from %p

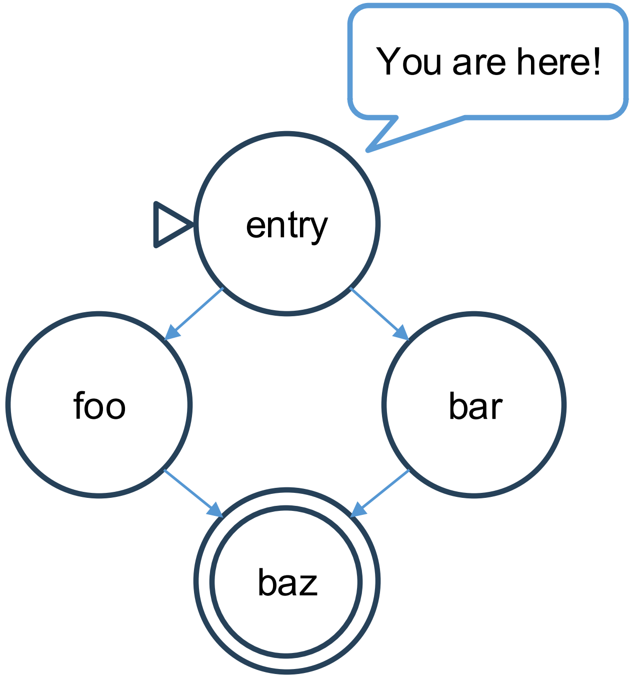
```
define void @f(i1 %c, ptr %p, ptr %q) {
entry:
  %x0 = load i64, ptr %p
  %p1 = getelementptr i64, ptr %p, i64 1
  %x1 = load i64, ptr %p1
  br i1 %c, label %foo, label %bar
foo:
  call void @g()
  br label %baz
bar:
  call void @g()
  br label %baz
baz:
  store i64 %x0, ptr %q
  %q1 = getelementptr i64, ptr %q, i64 1
  store i64 %x1, ptr %q1
  ret void
}
```



The 544.nab_r case

```
define void @f(i1 %c, ptr %p, ptr %q) {
entry:
  %x0 = load i64, ptr %p
  %p1 = getelementptr i64, ptr %p, i64 1
  %x1 = load i64, ptr %p1
  br i1 %c, label %foo, label %bar
foo:
  call void @g()
  br label %baz
bar:
  call void @g()
  br label %baz
baz:
  store i64 %x0, ptr %q
  %q1 = getelementptr i64, ptr %q, i64 1
  store i64 %x1, ptr %q1
  ret void
}
```

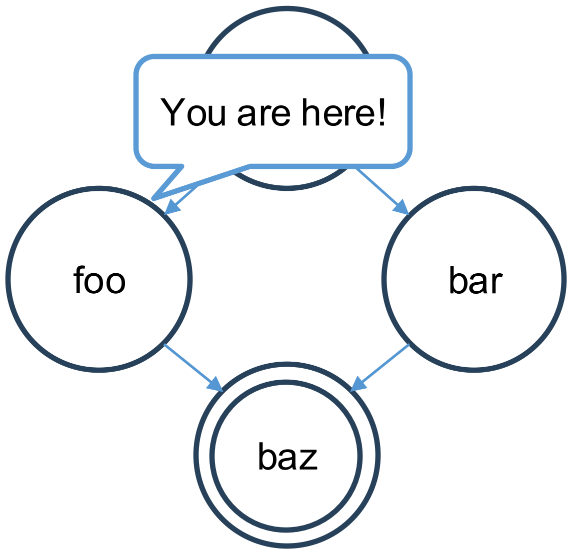
Conditional jump



The 544.nab_r case


```
define void @f(i1 %c, ptr %p, ptr %q) {
entry:
  %x0 = load i64, ptr %p
  %p1 = getelementptr i64, ptr %p, i64 1
  %x1 = load i64, ptr %p1
  br i1 %c, label %foo, label %bar
foo:
  call void @g()
  br label %baz
bar:
  call void @g()
  br label %baz
baz:
  store i64 %x0, ptr %q
  %q1 = getelementptr i64, ptr %q, i64 1
  store i64 %x1, ptr %q1
  ret void
}
```

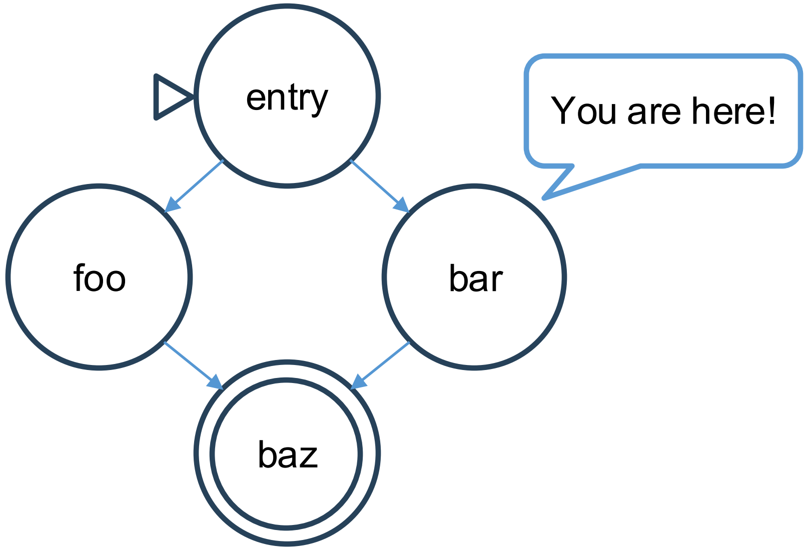
Conditional call to g()



The 544.nab_r case


```
define void @f(i1 %c, ptr %p, ptr %q) {
entry:
  %x0 = load i64, ptr %p
  %p1 = getelementptr i64, ptr %p, i64 1
  %x1 = load i64, ptr %p1
  br i1 %c, label %foo, label %bar
foo:
  call void @g()
  br label %baz
bar:
  call void @g()
  br label %baz
baz:
  store i64 %x0, ptr %q
  %q1 = getelementptr i64, ptr %q, i64 1
  store i64 %x1, ptr %q1
  ret void
}
```

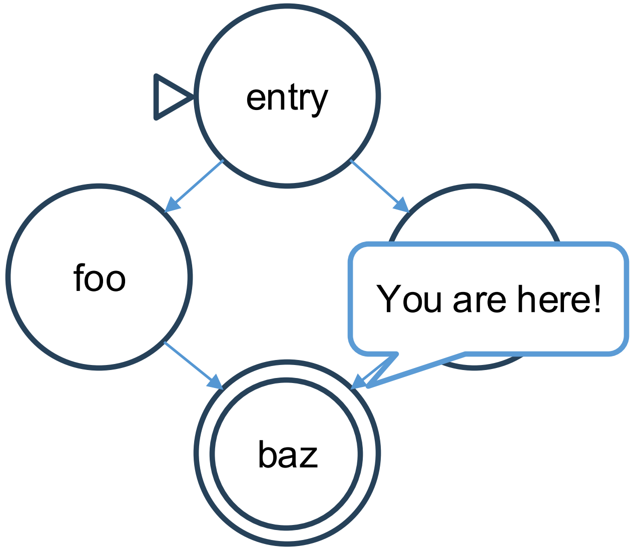
Conditional call to g()



The 544.nab_r case

```
define void @f(i1 %c, ptr %p, ptr %q) {
entry:
  %x0 = load i64, ptr %p
  %p1 = getelementptr i64, ptr %p, i64 1
  %x1 = load i64, ptr %p1
  br i1 %c, label %foo, label %bar
foo:
  call void @g()
  br label %baz
bar:
  call void @g()
  br label %baz
baz:
  store i64 %x0, ptr %q
  %q1 = getelementptr i64, ptr %q, i64 1
  store i64 %x1, ptr %q1
  ret void
}
```

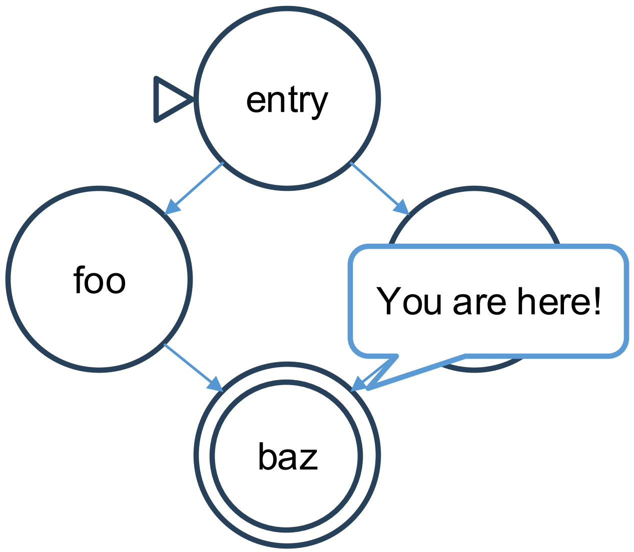
Stores first value from %q



The 544.nab_r case

```
define void @f(i1 %c, ptr %p, ptr %q) {
entry:
  %x0 = load i64, ptr %p
  %p1 = getelementptr i64, ptr %p, i64 1
  %x1 = load i64, ptr %p1
  br i1 %c, label %foo, label %bar
foo:
  call void @g()
  br label %baz
bar:
  call void @g()
  br label %baz
baz:
  store i64 %x0, ptr %q
  %q1 = getelementptr i64, ptr %q, i64 1
  store i64 %x1, ptr %q1
  ret void
}
```

Stores second value from %q



Fixing Real Bugs

- Since we are storing and loading from continuous regions, the accesses can be vectorized.

Fixing Real Bugs

- Since we are storing and loading from continuous regions, the accesses can be vectorized.
- We found that the SLP Vectorizer was aggressively vectorizing regions without properly accounting for the cost of spilling vector registers across calls.

Fixing Real Bugs

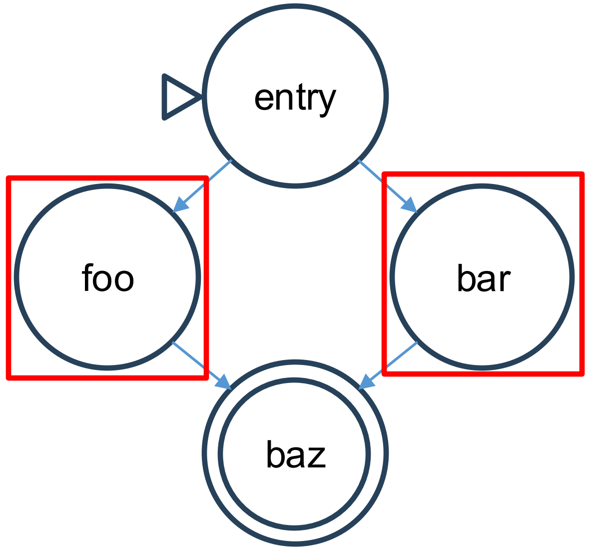
RISC-V SUMMIT NORTH AMERICA 2025

- Since we are storing and loading from continuous regions, the accesses can be vectorized.
- We found that the SLP Vectorizer was aggressively vectorizing regions without properly accounting for the cost of spilling vector registers across calls.
- Previously, the SLP vectorizer only analyzed the entry and baz blocks, ignoring foo and bar entirely.

The 544.nab_r case

Not being analyzed

```
define void @f(i1 %c, ptr %p, ptr %q) {
entry:
  %x0 = load i64, ptr %p
  %p1 = getelementptr i64, ptr %p, i64 1
  %x1 = load i64, ptr %p1
  br i1 %c, label %foo, label %bar
foo:
  call void @g()
  br label %baz
bar:
  call void @g()
  br label %baz
baz:
  store i64 %x0, ptr %q
  %q1 = getelementptr i64, ptr %q, i64 1
  store i64 %x1, ptr %q1
  ret void
}
```



Addressing the Issue

- To address the issue, we first proposed a patch which modified the SLP vectorizer to properly walk through all basic blocks when analyzing cost.

Addressing the Issue

- To address the issue, we first proposed a patch which modified the SLP vectorizer to properly walk through all basic blocks when analyzing cost.
- Promising results: execution time dropped by **9.92%** in 544.nab_r.

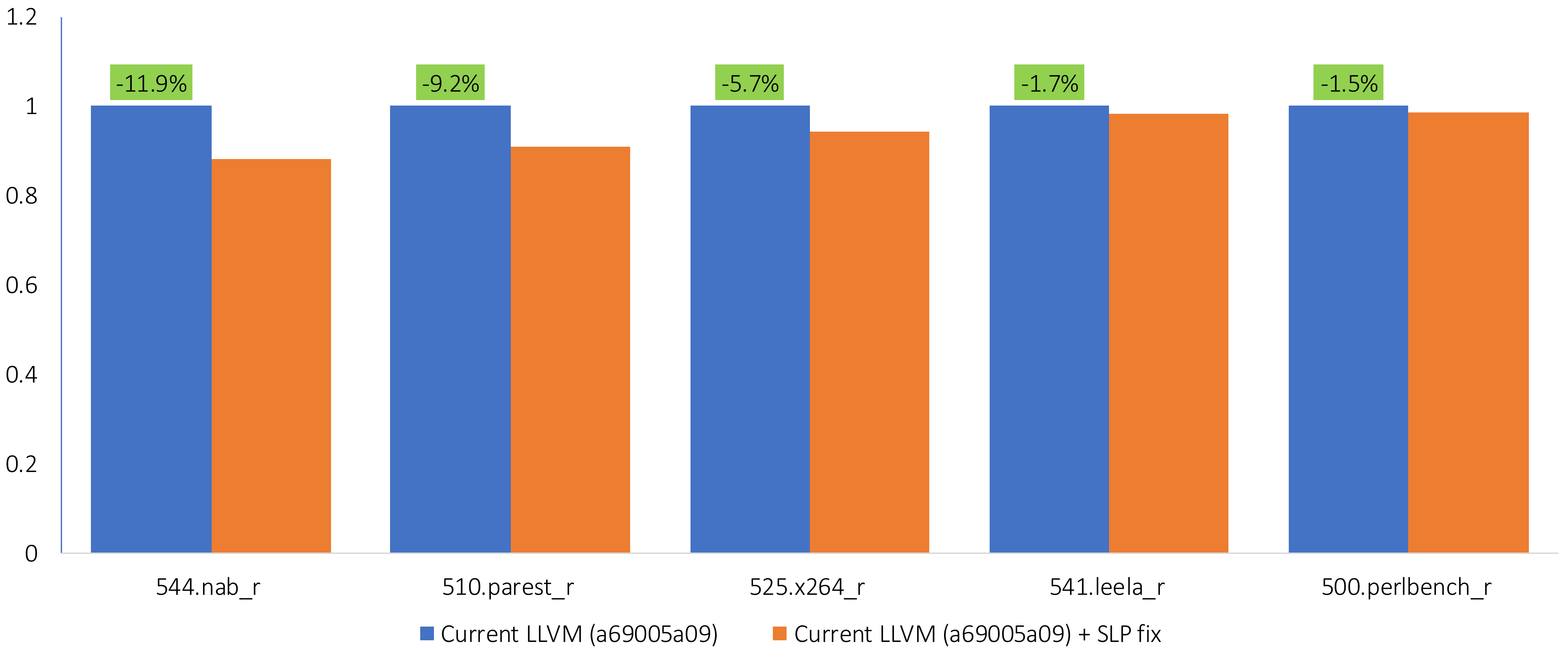
Addressing the Issue

- To address the issue, we first proposed a patch which modified the SLP vectorizer to properly walk through all basic blocks when analyzing cost.
- Promising results: execution time dropped by **9.92%** in 544.nab_r.
- But with a major drawback: **+6.9%** increase compilation time in 502.gcc_r.

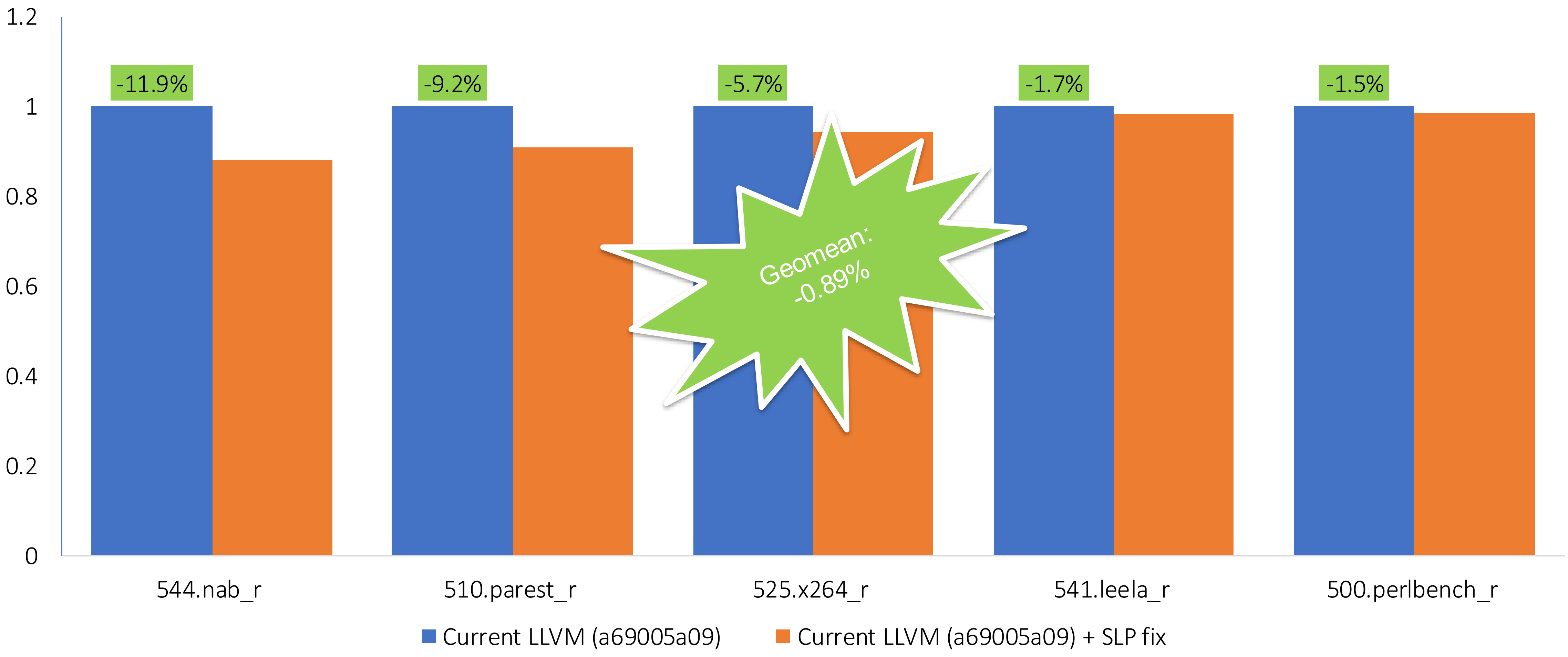
Addressing the Issue

- To address the issue, we first proposed a patch which modified the SLP vectorizer to properly walk through all basic blocks when analyzing cost.
- Promising results: execution time dropped by **9.92%** in 544.nab_r.
- But with a major drawback: **+6.9%** increase compilation time in 502.gcc_r.
- Following discussions with the community, Alexey Bataev (SLP Vectorizer code owner) proposed and landed refined solution.

RVA22U64_V SPEC exec time, O3+LTO, SLP fix



RVA22U64_V SPEC exec time, O3+LTO, SLP fix



IPRA (Inter-Procedural Register Allocation) Support

IPRA (Inter-Procedural Register Allocation) Support

- Function calls can often spill (when you store a register to the stack) more registers than necessary → wasted cycles on every call.

IPRA (Inter-Procedural Register Allocation) Support

- Function calls can often spill (when you store a register to the stack) more registers than necessary → wasted cycles on every call.
- IPRA: caller/callee register use is tracked across the functions, by eliminating unnecessary save/restore sequences.

What IPRA Brings

```
foo:
        addi    sp, sp, -32
        sd      ra, 24(sp)
        sd      s0, 16(sp)
        sd      s1, 8(sp)
        ...
        ld      s1, 8(sp)
        ld      s0, 16(sp)
        ld      ra, 24(sp)
        addi    sp, sp, 32
        ret
```

before

```
foo:
        addi    sp, sp, -8
        sd      ra, 0(sp)
        ...
        ld      ra, 0(sp)
        addi    sp, sp, 8
        ret
```

after

What IPRA Brings


```
foo:
      addi    sp, sp, -3
      sd      ra, 24(sp)
      sd      s0, 16(sp)
      sd      s1, 8(sp)
      ...
      ld      s1, 8(sp)
      ld      s0, 16(sp)
      ld      ra, 24(sp)
      addi    sp, sp, 32
      ret
```

saved even if not really live

saved even if not really live

before

```
foo:
      addi    sp, sp, -8
      sd      ra, 0(sp)
      ...
      ld      ra, 0(sp)
      addi    sp, sp, 8
      ret
```

after

What IPRA Brings

```
foo:
        addi    sp, sp, -32
        sd      ra, 24(sp)
        sd      s0, 16(sp)
        sd      s1, 8(sp)
        ...
        ld      s1, 8(sp)
        ld      s0, 16(sp)
        ld      ra, 24(sp)
        addi    sp, sp, 32
        ret
```

before

```
foo:
        addi    sp, sp, -8
        sd      ra, 0(sp)
        ...
        ld      ra, 0(sp)
        addi    sp, sp, 8
        ret
```

only what's truly needed

after

Impact of IPRA

- Reduction in register pressure, shorter prologue/epilogue code.

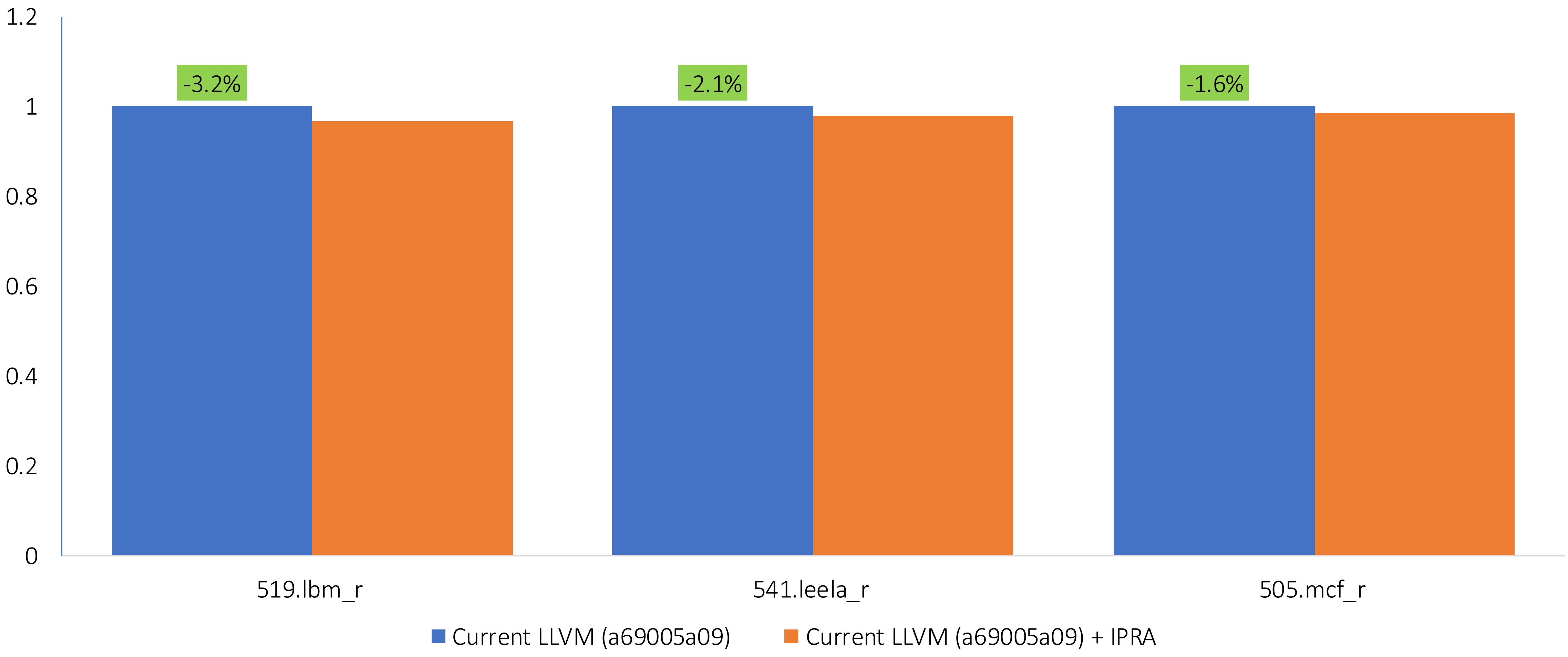
Impact of IPRA

- Reduction in register pressure, shorter prologue/epilogue code.
- SPEC benchmarks showed measurable improvements (small but consistent).

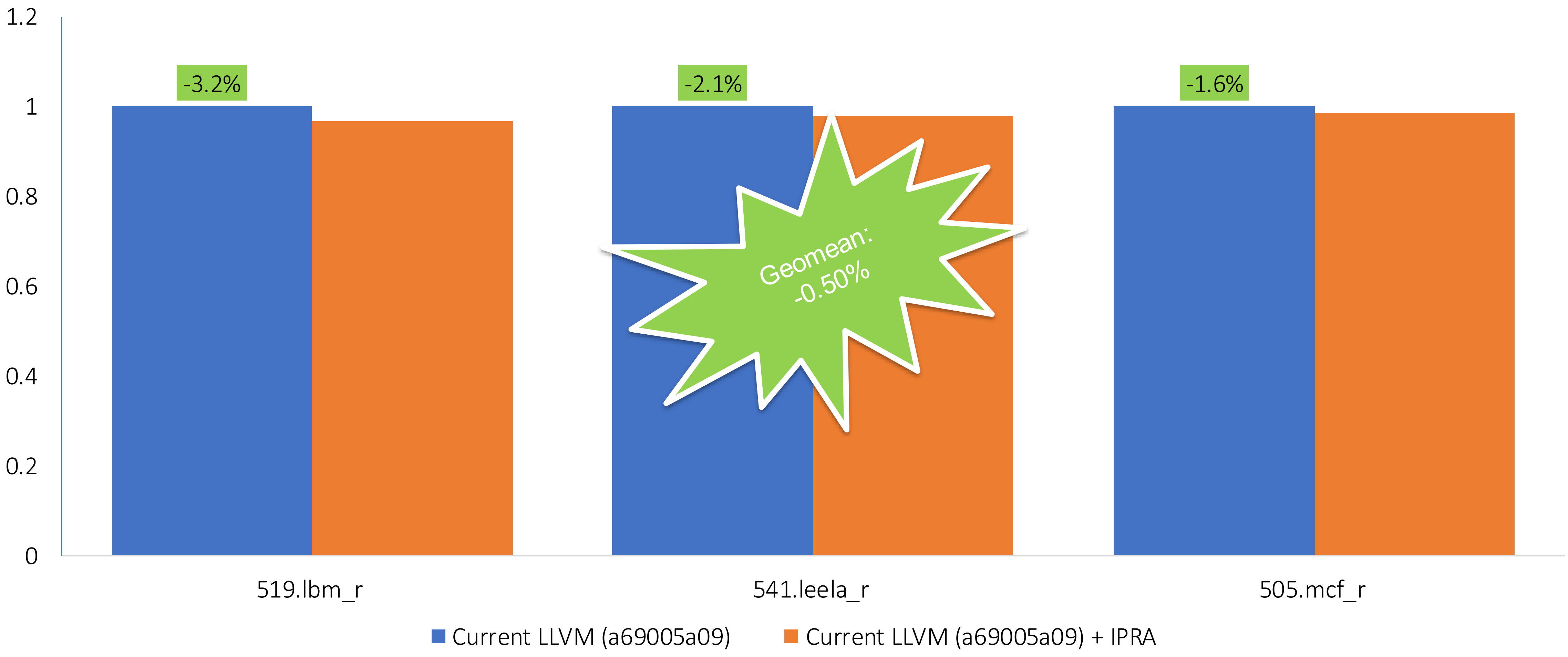
Impact of IPRA

- Reduction in register pressure, shorter prologue/epilogue code.
- SPEC benchmarks showed measurable improvements (small but consistent).
- Unfortunately, it can’t be enabled by default: IPRA is not enabled by default due to a bug (described in issue 119556), however, it does not affect the SPEC benchmarks.

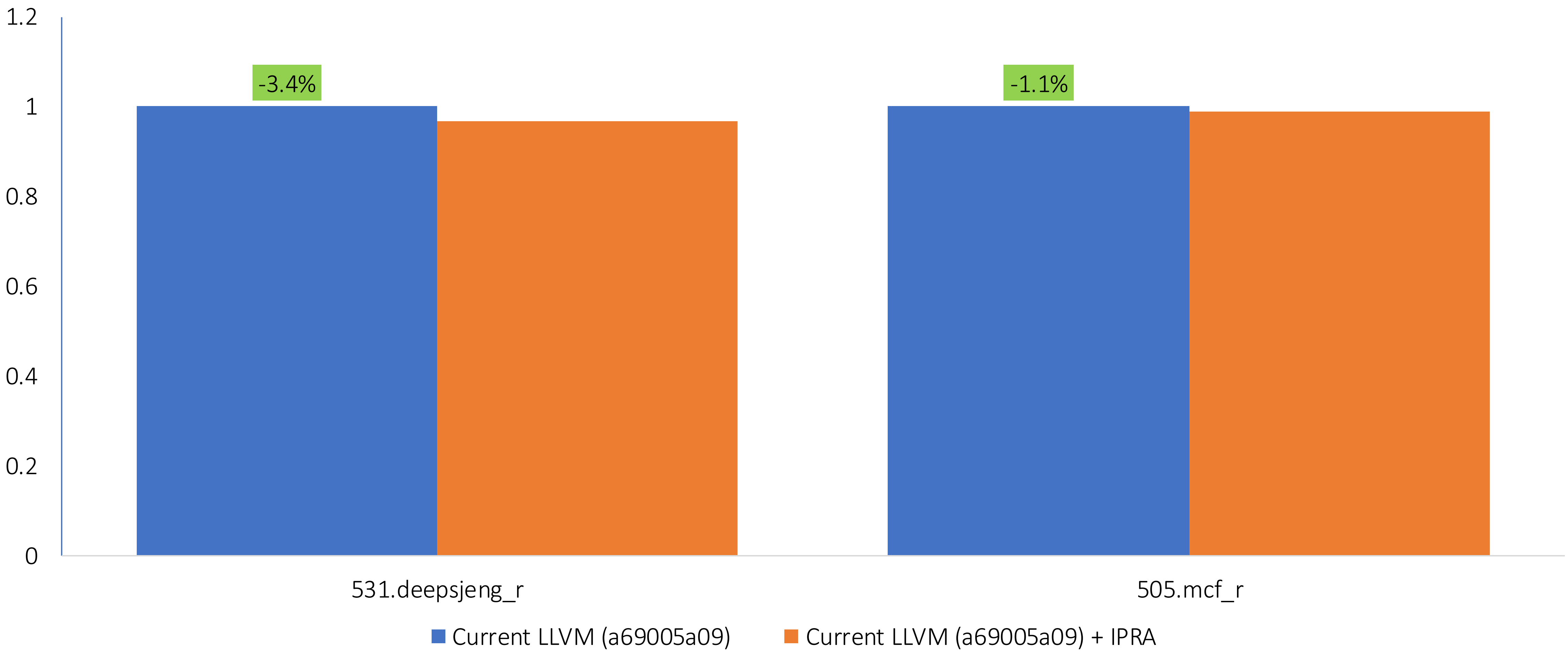
RVA22U64 SPEC exec time, O3+LTO+IPRA



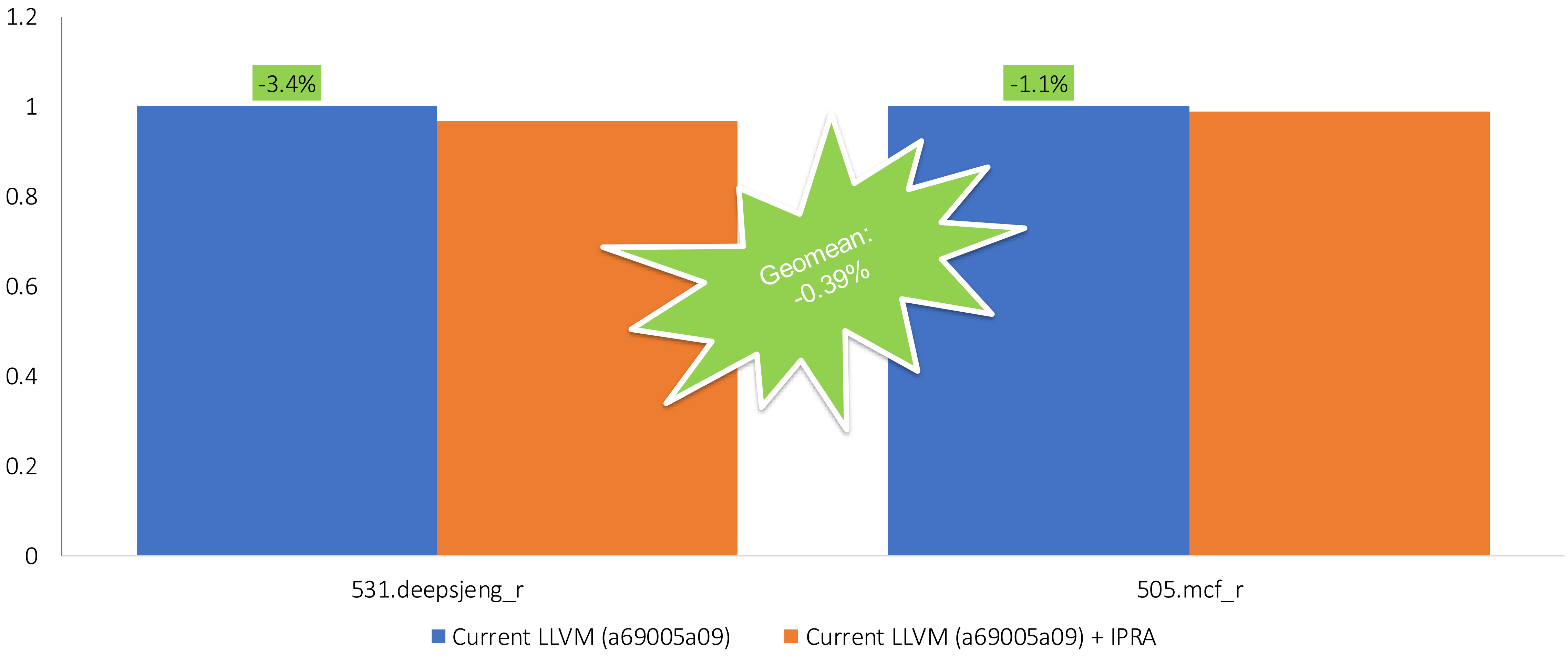
RVA22U64 SPEC exec time, O3+LTO+IPRA



RVA22U64_V SPEC exec time, O3+LTO+IPRA



RVA22U64_V SPEC exec time, O3+LTO+IPRA



Conclusions

Faster RISC-V Today

- We contributed with:
  - Scheduling: largest wins, especially for scalar-heavy code.
  - Vectorization: enabled smarter spilling cost calculations.
  - IPRA: smaller but consistent improvements across workloads.

Faster RISC-V Today

RISC-V SUMMIT NORTH AMERICA 2025

- We contributed with:
  - Scheduling: largest wins, especially for scalar-heavy code.
  - Vectorization: enabled smarter spilling cost calculations.
  - IPRA: smaller but consistent improvements across workloads.
- Almost all changes are upstream benefiting everyone. Under review:
  - https://github.com/llvm/llvm-project/pull/150618
  - https://github.com/llvm/llvm-project/pull/150644
  - https://github.com/llvm/llvm-project/pull/152557
  - https://github.com/llvm/llvm-project/pull/152737
  - https://github.com/llvm/llvm-project/pull/152738

What We Learned Along the Way

- Scheduling is critical for performance.
  - No scheduling model → LLVM pessimises the final code.
  - We should likely adopt some scheduling model as default, like other targets do.
  - Should we make the X60 scheduling model default for in-order RISC-V processors?

What We Learned Along the Way

- Scheduling is critical for performance.
  - No scheduling model → LLVM pessimises the final code.
  - We should likely adopt some scheduling model as default, like other targets do.
  - Should we make the X60 scheduling model default for in-order RISC-V processors?
- Many contributions don’t have immediate benchmark impact.

What We Learned Along the Way

- Scheduling is critical for performance.
  - No scheduling model → LLVM pessimizes the final code.
  - We should likely adopt some scheduling model as default, like other targets do.
  - Should we make the X60 scheduling model default for in-order RISC-V processors?
- Many contributions don't have immediate benchmark impact.
- Vectorization needs careful tuning to avoid regressions.

Did we close the performance gap between LLVM and the GCC compiler?

LLVM vs GCC on SpacemiT-X60

- **Note it’s not a direct apples-to-apples comparison.**

LLVM vs GCC on SpacemiT-X60

- **Note it’s not a direct apples-to-apples comparison**.
- The code was compiled with the same RISC-V extensions enabled.

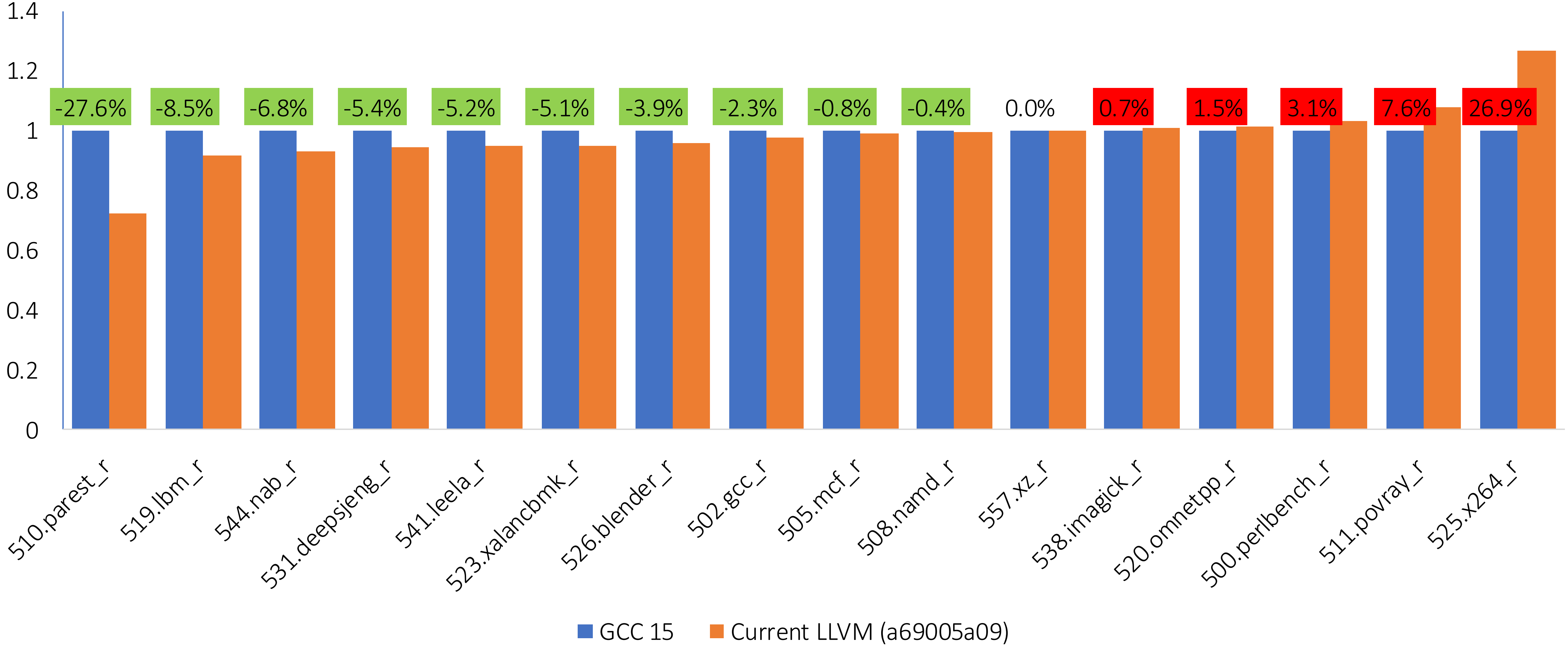
LLVM vs GCC on SpacemiT-X60

- **Note it’s not a direct apples-to-apples comparison**.
- The code was compiled with the same RISC-V extensions enabled.
- But GCC doesn’t have X60-specific scheduling latencies, while LLVM does.

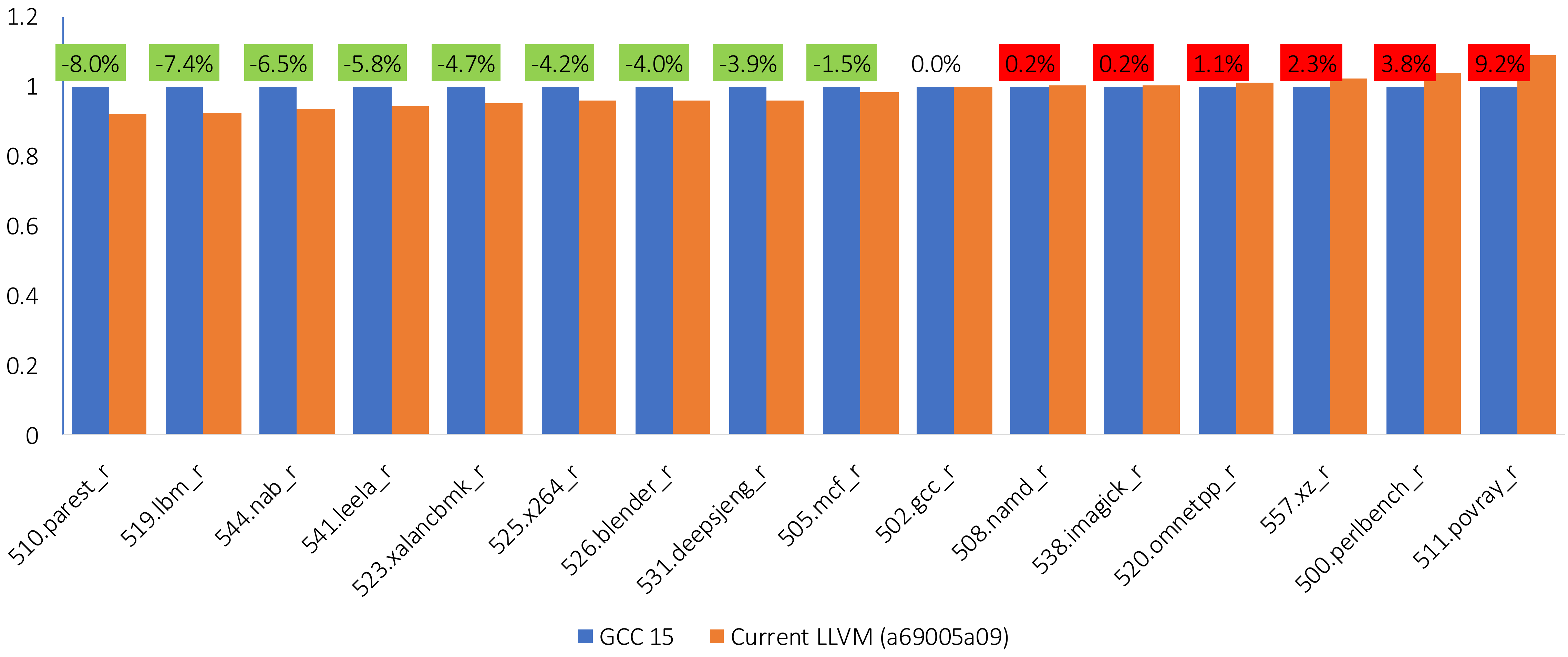
LLVM vs GCC on SpacemiT-X60

- **Note it’s not a direct apples-to-apples comparison**.
- The code was compiled with the same RISC-V extensions enabled.
- But GCC doesn’t have X60-specific scheduling latencies, while LLVM does.
- Still useful to show relative progress and identify where LLVM has caught up.

RVA22U64 SPEC execution time, GCC vs LLVM



RVA22U64_V SPEC execution time, GCC vs LLVM



Thanks!

- This work at Igalia was made possible thanks to support from RISE, under Project RP009.
- Thanks to my Igalia colleagues for discussions, and feedback.
- And to the LLVM RISC-V community for reviews and for getting patches upstream quickly.

RISC-V @ Igalia

- Accidental Dataflow Analysis: Extending the RISC-V VL Optimizer @ 2025 EuroLLVM by Luke Lau: https://www.youtube.com/watch?v=bkOwPr36SrQ
- Improvements to RISC-V Vector code generation in LLVM @ 2025 RISC-V Summit Europe by Luke Lau and Alex Bradbury: https://www.youtube.com/watch?v=0NjugW7FF48
- RISC-V nightly performance testing of top-of-tree GCC and clang:

  https://cc-perf.igalia.com/

RISC-V SUMMIT

NORTH AMERICA 2025