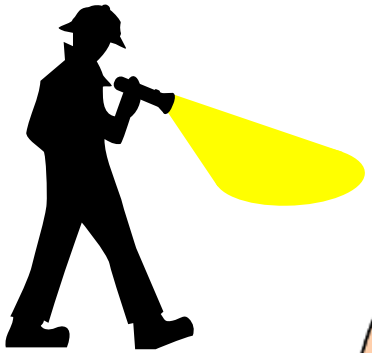
EMBECOSM®

How compact is compiled RISC-V code?

Jeremy Bennett

What this talk is...



Thanks to openclipart.org

What this talk is not...

Thanks to openclipart.org

EMBECOSM®

Architectures Analysed

- DesignWare ARC HS.

- Arm Cortex-M4 + Thumb 2.

- RISV-V RV32IMC

All 32-bit architectures, with 16-bit short instructions and no hardware floating point.

BEEBS

Name	B	M	I	F
Blowfish				
CRC32				
Cubic root solver				
Dijkstra				
FDCT				
Float matmult				
Integer matmult				
Rjindael				
SHA				
2D FIR				

High frequency

Medium frequency

Low frequency

Bristol/Embecosm Embedded Benchmark Suite

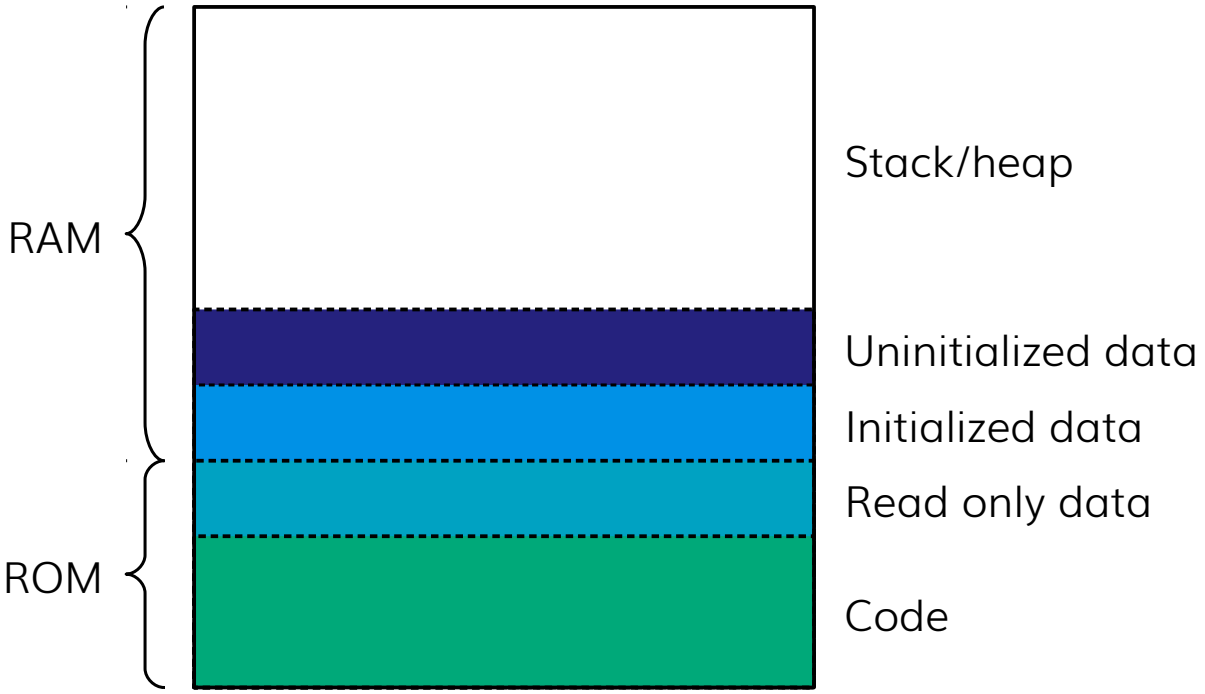
- free and open source
- mixture of branching (B), memory access (M), integer ops (I) and floating point (F)
- minimal I/O
- https://arxiv.org/abs/1308.5174
- BEEBS 2.0 now 79 benchmarks

EMBECOSM®

What to Measure

- Sections in an embedded program
  - **code:** goes into ROM/Flash
  - **read-only data:** goes into ROM/Flash
  - **initialized data:** goes into RAM, maybe setup from ROM
  - **uninitialized data (BSS)**: goes into RAM
- We look at **code** + **read-only data** size
  - most important for embedded systems
  - easily measured using `size`

What to Measure



Look at:

code + **read-only data**

- most important for embedded systems
- easily measured using `size`

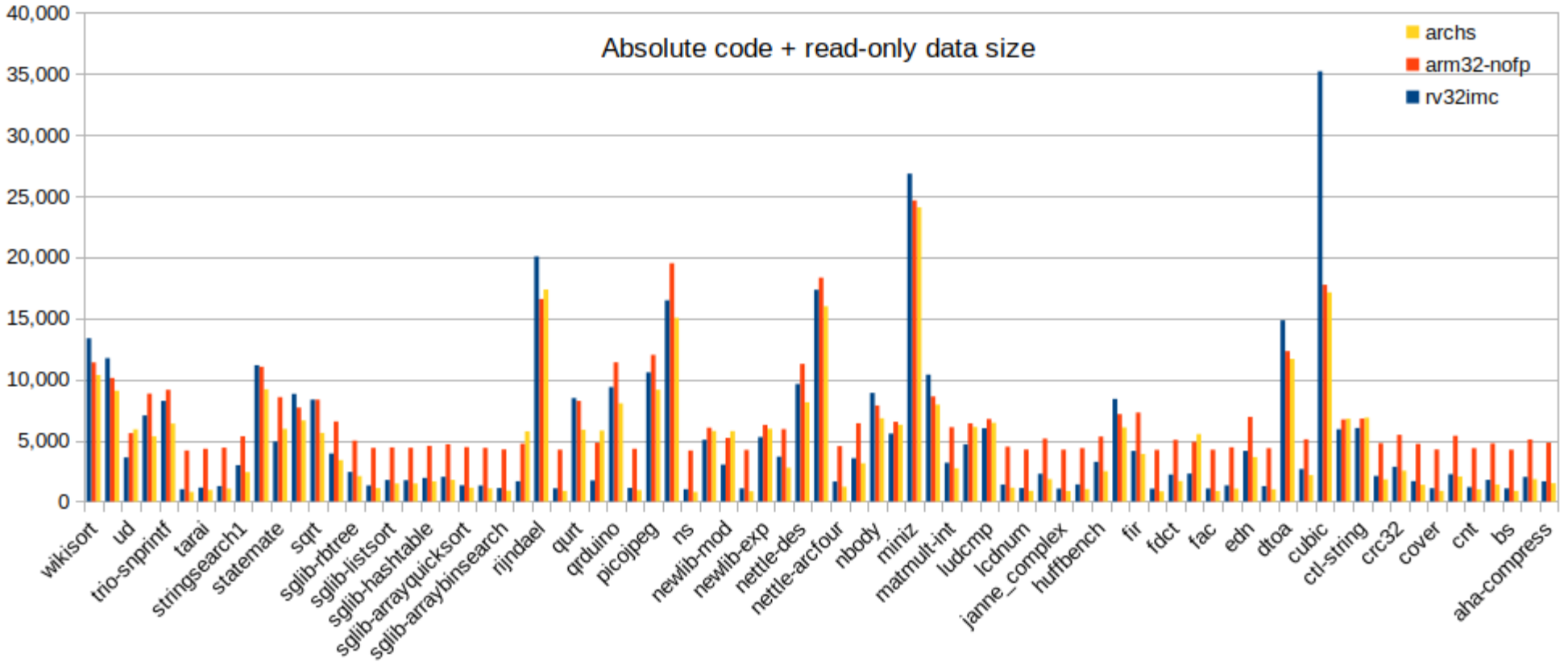
Absolute Statistics

- Size for each program
  - code + read-only data
- Statistics recorded:
  - Total size for all programs
    - dominated by effects on large programs
  - Size of largest program
  - Size of smallest programs

Relative Statistics

- Size for ARC and Arm against RISC-V as baseline
  - choice of baseline is arbitrary
- Statistics recorded:
  - relative size for each program
  - arithmetic average for all programs
    - should we have used geometric or harmonic mean?
  - smallest relative size
  - largest relative size

Baseline Results



EMBECOSM®

Baseline Summary

	ARC absolute	ARC relative	Arm absolute	Arm relative	RISC-V absolute	RISC-C relative
Total/average	357,058		543,290		407,693	
Minimum	780		4,168		994	
Maximum	24,068		24,638		35,168	

Baseline Summary

	ARC absolute	ARC relative	Arm absolute	Arm relative	RISC-V absolute	RISC-C relative
Total/average	357,058	96%	543,290	222%	407,693	100%
Minimum	780	49%	4,168	50%	994	100%
Maximum	24,068	347%	24,638	419%	35,168	100%

Baseline Summary

	ARC absolute	ARC relative	Arm absolute	Arm relative	RISC-V absolute	RISC-C relative
Total/average	357,058	96%	543,290	222%	407,693	100%
Minimum	780	49%	4,168	50%	994	100%
Maximum	24,068	347%	24,638	419%	35,168	100%

- Why does Arm do so badly with small programs?

The Smallest Program, `ns`

```
$ arc-elf32-nm src/ns/ns
00000294 T benchmark
00002514 G __bss_start
000002c8 T __call_exitprocs
00002514 b completed.3536
…
00000368 T __st_r13_to_r24
00000364 T __st_r13_to_r25
00002500 D __TMC_END__
0000028c T verify_benchmark
```

61 symbols defined

```
$ arm-none-eabi-nm src/ns/ns
0000907c r
000081ae T atexit
000081a4 T benchmark
00019364 B __bss_end__
…
0000819c T verify_benchmark
000082d4 t wrap.part.1
00008466 W _write
00008c9c T _write_r
```

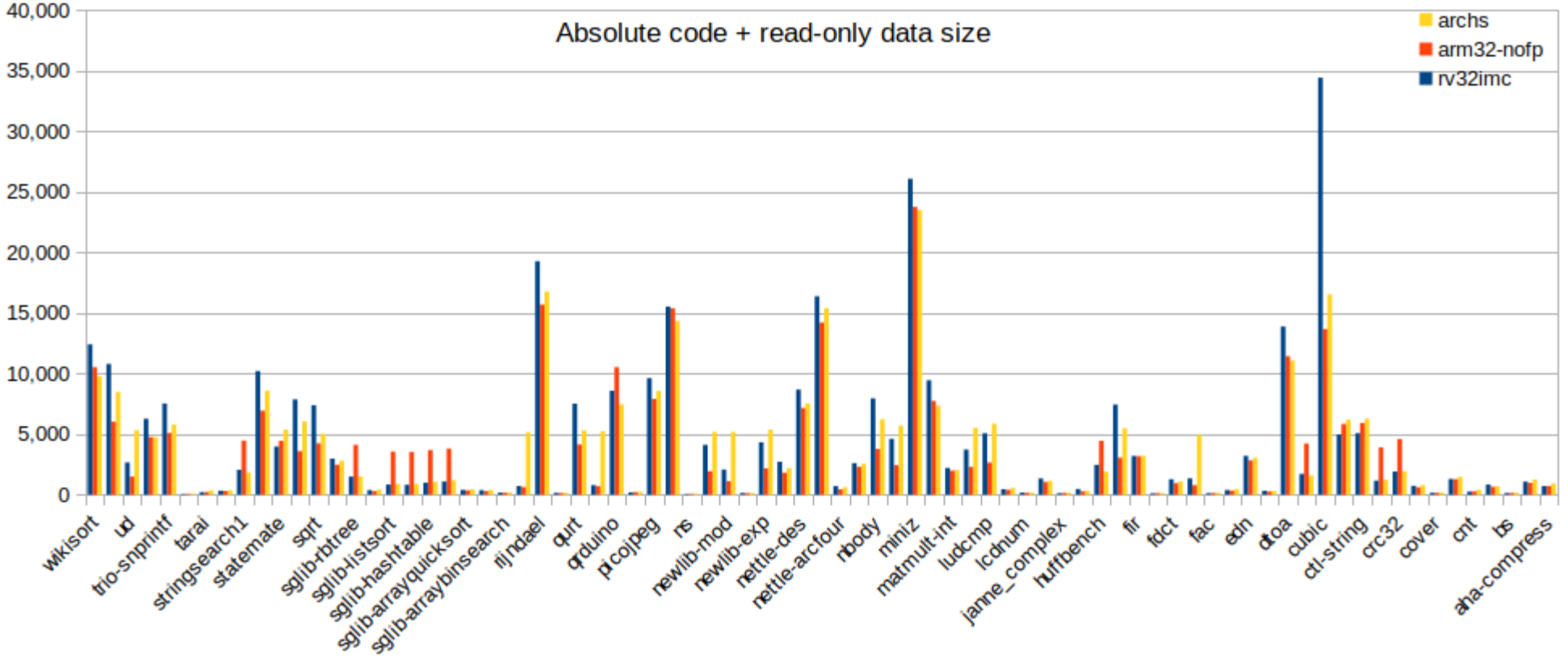
130 symbols defined

```
$ riscv32-unknown-elf-nm src/ns/ns
00011478 d
00010196 T atexit
00010186 T benchmark
0001157c B __bss_start
…
0001018e T start_trigger
00010192 T stop_trigger
00011574 G __TMC_END__
00010180 T verify_benchmark
```

43 symbols defined

- Arm is pulling in a lot of standard C library code
- Culprit is the C runtime startup, **`crt0.o`**

Results with Dummy `crt0.o`



EMBECOSM®

The Smallest Program, `ns`, Without `crt0.o`

```
$ arc-elf32-nm src/ns/ns
00000154 T benchmark
00002164 T __bss_start
00002164 T _edata
00002164 T _end
…
00002164 B __start_heap
0000015c T start_trigger
00000160 T stop_trigger
0000014c T verify_benchmar
```

18 symbols defined

```
$ arm-none-eabi-nm src/ns/ns
00008044 T benchmark
0001804e T __bss_end__
0001804e T _bss_end__
0001804e T __bss_start
…
00008034 T _start
0000804a T start_trigger
0000804c T stop_trigger
0000803c T verify_benchmark
```

17 symbols defined

```
$ riscv32-unknown-elf-nm src/ns/ns
0001008c T benchmark
0001109c T __bss_start
0001109c T _edata
0001109c T _end
…
00010080 T _start
00010094 T start_trigger
00010098 T stop_trigger
00010086 T verify_benchmark
```

12 symbols defined

- The playing field is leveled for comparison

Dummy `crt0.o` Summary

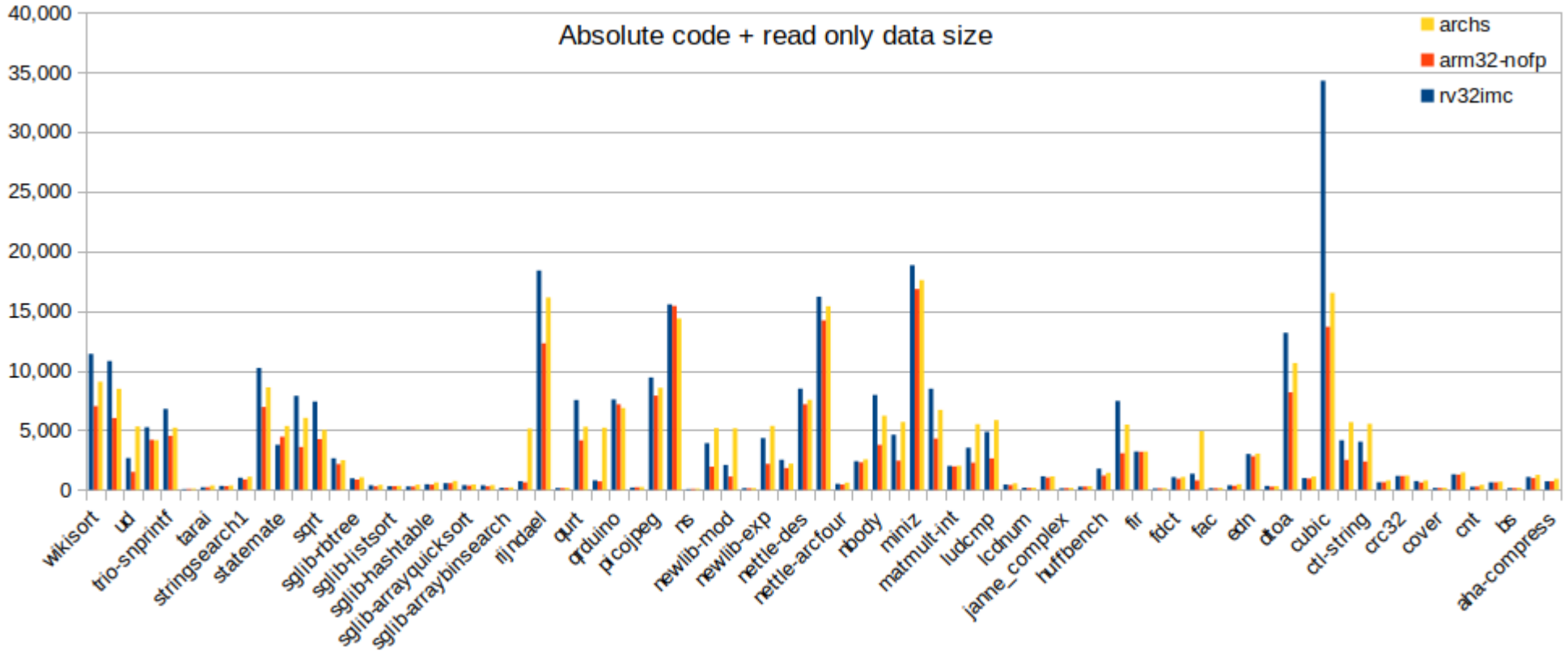
	ARC absolute	ARC relative	Arm absolute	Arm relative	RISC-V absolute	RISC-C relative
Total/average	309,387	125%	278,598	111%	335,673	100%
Minimum	100	48%	78	40%	66	100%
Maximum	23,504	715%	23,770	433%	34,442	100%

Dummy `crt0.o` Summary

	ARC absolute	ARC relative	Arm absolute	Arm relative	RISC-V absolute	RISC-C relative
Total/average	309,387	125%	278,598	111%	335,673	100%
Minimum	100	48%	78	40%	66	100%
Maximum	23,504	715%	23,770	433%	34,442	100%

- Some of the larger programs do use the C library
  - why are these programs often larger for ARC and Arm?

Results without Standard C Library



EMBECOSM®

No Standard C Library Summary

	ARC absolute	ARC relative	Arm absolute	Arm relative	RISC-V absolute	RISC-C relative
Total/average	291,735	130%	212,839	83%	309,774	100%
Minimum	100	48%	78	40%	66	100%
Maximum	17,572	712%	16,822	118%	34,272	100%

- Note that the pathological Arm cases have gone
  - but not ARC

Absolute Effect of No Standard C Library

	ARC	Arm	RISC-V
Total no `crt0.o`	309,387	278,598	335,673
Total no `crt0.o` or `libc`	291,735	212,839	309,774
Difference (absolute)	17,652	65,759	25,899
Difference (relative)	6%	24%	8%

- ARM standard C library is *much* larger
  - prioritize performance over size?
  - no multilib variants for `-Os`?
  - multiple functions per object file and no `-gc-sections`

Some Notable Variations

	ARC	Arm	RISC-V
`cubic`	48%	40%	100%
`frac`	73%	41%	100%

Some Notable Variations

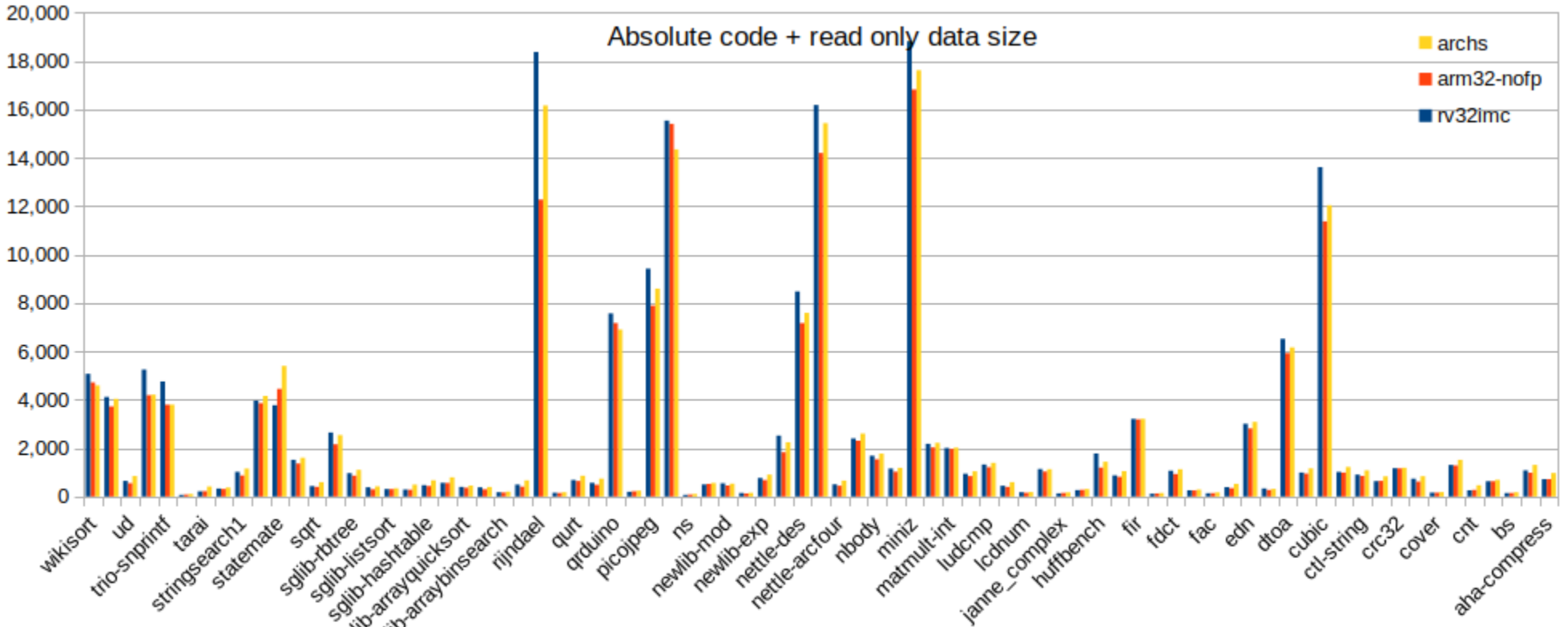
	ARC	Arm	RISC-V
`cubic`	48%	40%	100%
`frac`	73%	41%	100%
`matmult-float`	156%	64%	100%
`matmult-int`	101%	98%	100%

Some Notable Variations

	ARC	Arm	RISC-V
`cubic`	48%	40%	100%
`frac`	73%	41%	100%
`matmult-float`	156%	64%	100%
`matmult-int`	101%	98%	100%

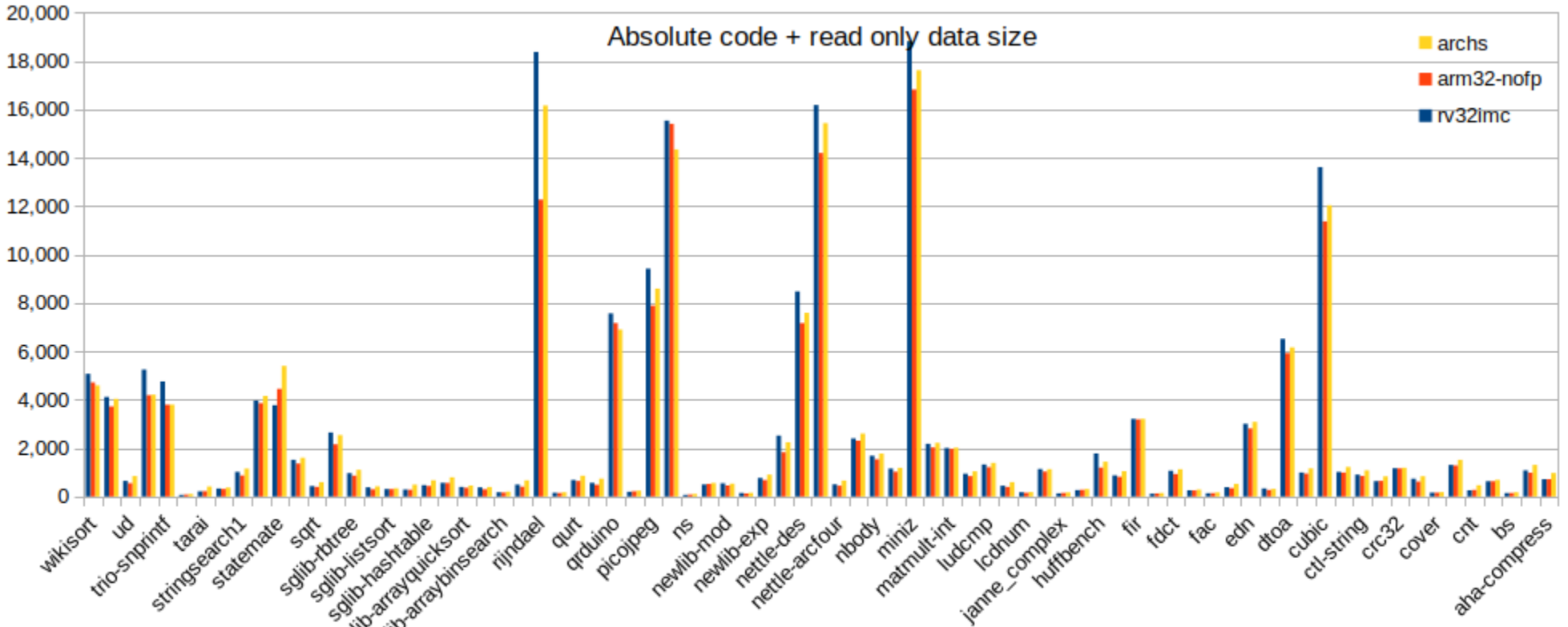
- Arm, seems to do a very good job with floating point
  - ARC more variable
  - these are all emulated floating point in `libgcc`

Results with Dummy Emulation Library



EMBECOSM®

Results with Dummy Emulation Library



EMBECOSM®

Dummy Emulation Library Summary

	ARC absolute	ARC relative	Arm absolute	Arm relative	RISC-V absolute	RISC-C relative
Total/average	188,787	115%	170,717	93%	192,961	100%
Minimum	66	80%	78	67%	66	100%
Maximum	18,820	194%	16,822	118%	18,820	100%

- Pathological ARC cases much improved

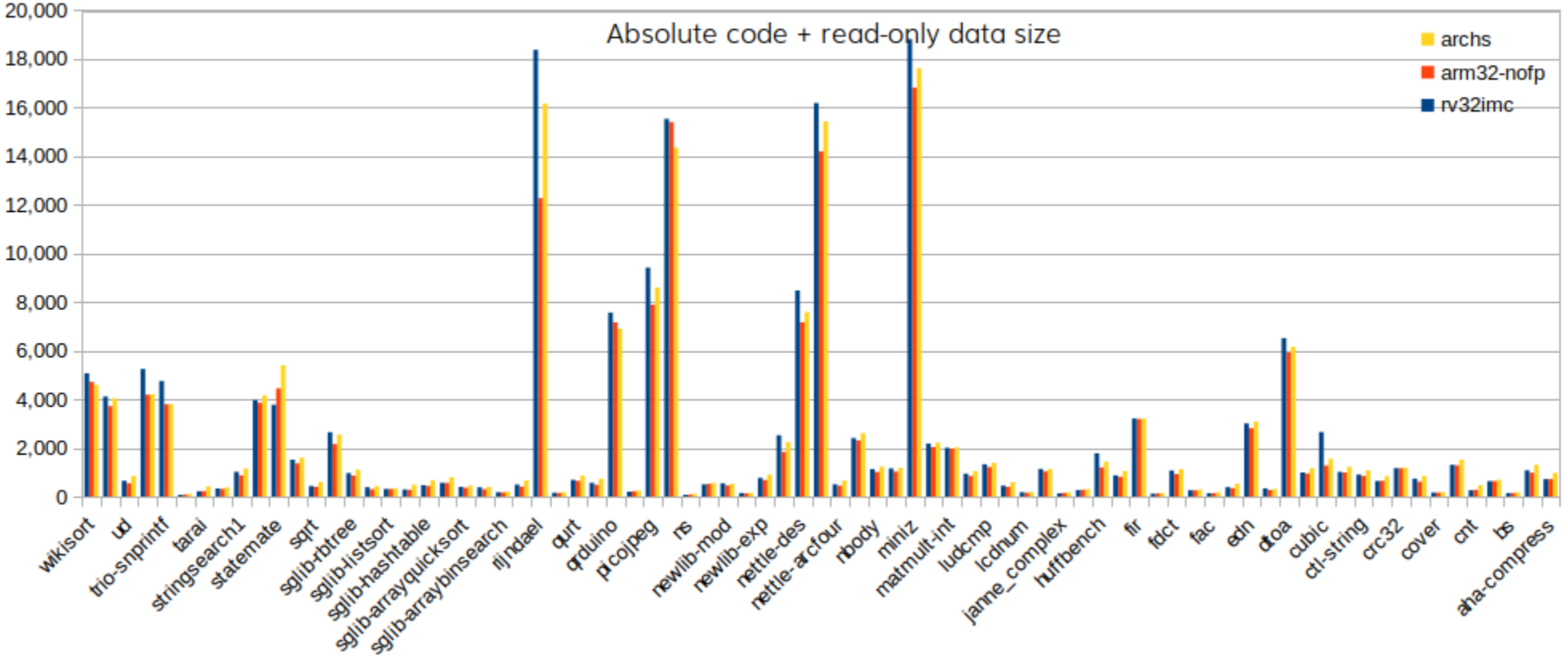
EMBECOSM®

Absolute Effect of No Emulation Library

	ARC	Arm	RISC-V
Total no **`crt0.o`** or **`libc`**	291,735	212,839	309,774
Total no **`crt0.o`**, **`libc`** or **`libgcc`**	188,787	170,717	193,961
Difference (absolute)	102,948	42,122	115,813
Difference (relative)	35%	20%	37%

- ARC and RISC-V emulation is over one third of code size
  - for ARM it is just one fifth
- But is it just floating point emulation
  - **`libgcc`** does much more than just floating point

Results without Math Library



EMBECOSM®

No Math Library Summary

	ARC absolute	ARC relative	Arm absolute	Arm relative	RISC-V absolute	RISC-C relative
Total/average	177,779	114%	160,092	92%	182,439	100%
Minimum	100	59%	78	48%	66	100%
Maximum	17,620	194%	16,822	118%	18,820	100%

- Virtually no effect

Absolute Effect of No Math Library

	ARC	Arm	RISC-V
Total no **`crt0.o`**, **`libc`** or **`libgcc`**	188,787	170,717	193,961
Total no **`crt0.o`**, **`libc`**, **`libgcc`** or **`libm`**	177,759	160,092	182,439
Difference (absolute)	11,028	10,625	11,522
Difference (relative)	6%	6%	6%

- The effect is small overall, because few programs use the math library
- All architectures just use generic C code for this library, hence similar sizes
- Will have an impact on the programs that do use the math library

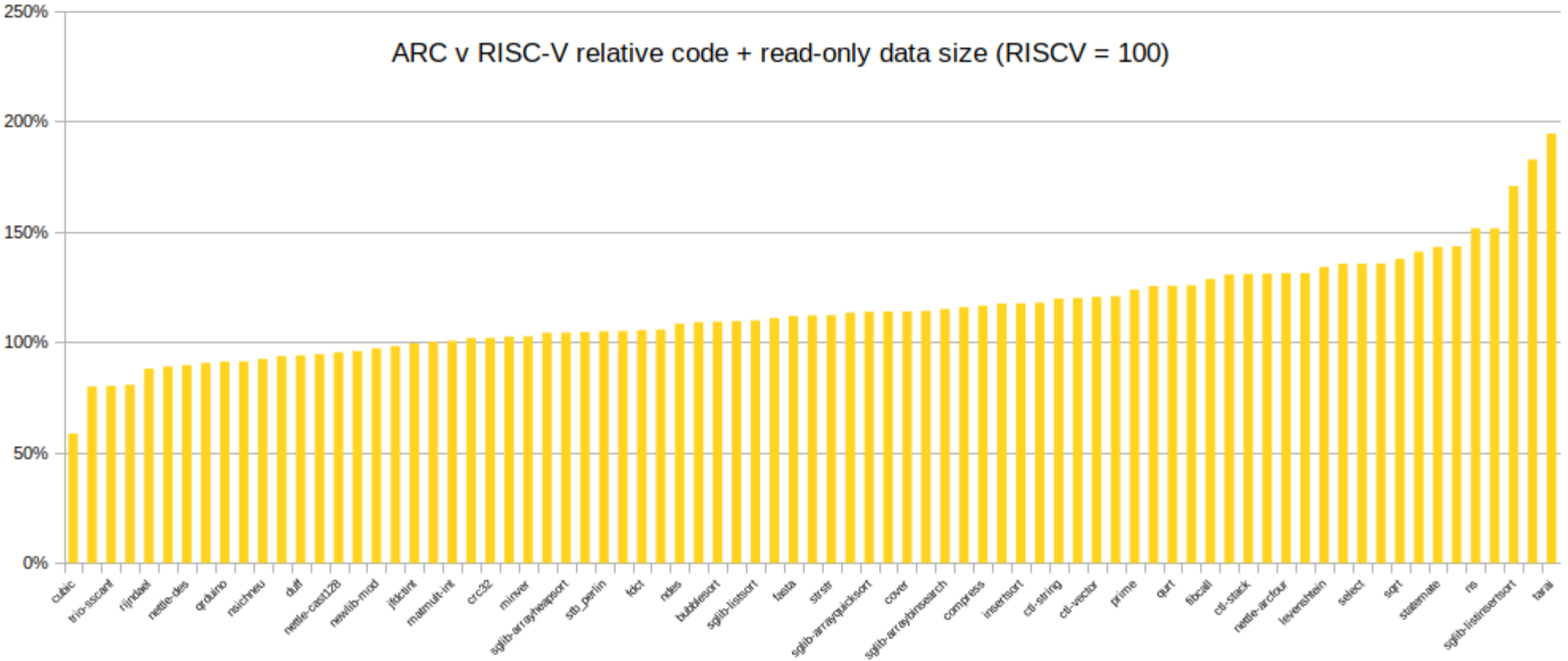
Overall Summary

	ARC absolute	ARC relative	Arm absolute	Arm relative	RISC-V absolute	RISC-C relative
Baseline	357,048	96%	543,290	222%	407,693	100%
+dummy crt0.o	309,387	125%	278,598	111%	335,673	100%
+dummy libc	291,735	130%	212,839	83%	309,774	100%
+dummy libgcc	188,787	115%	170,717	93%	192,961	100%
+dummy libm	177,779	114%	160,092	92%	182,439	100%

Takeaway

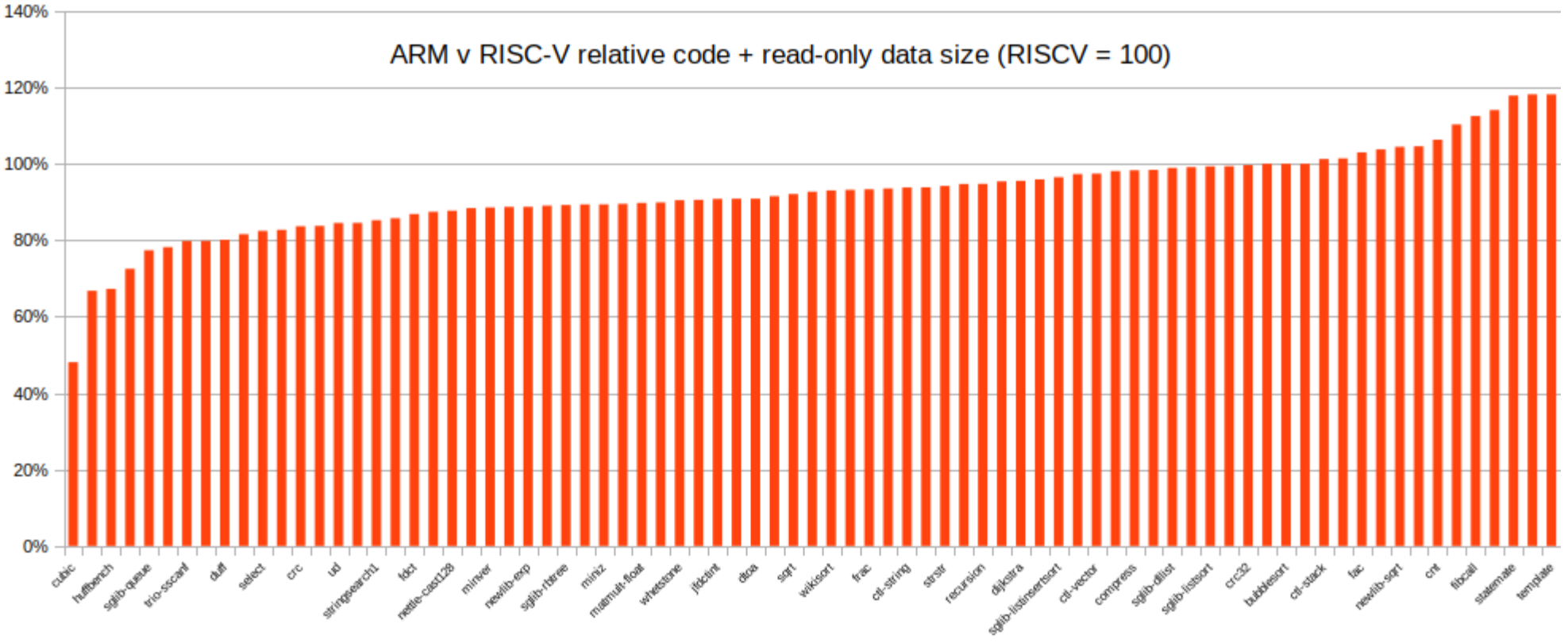
- *Useful* results only consider the code you compiled
  - libraries confound the results
- Therefore remove:
  - startup code
  - standard C library code
  - emulation library code
  - math library code

Useful Graph: ARC v RISC-V



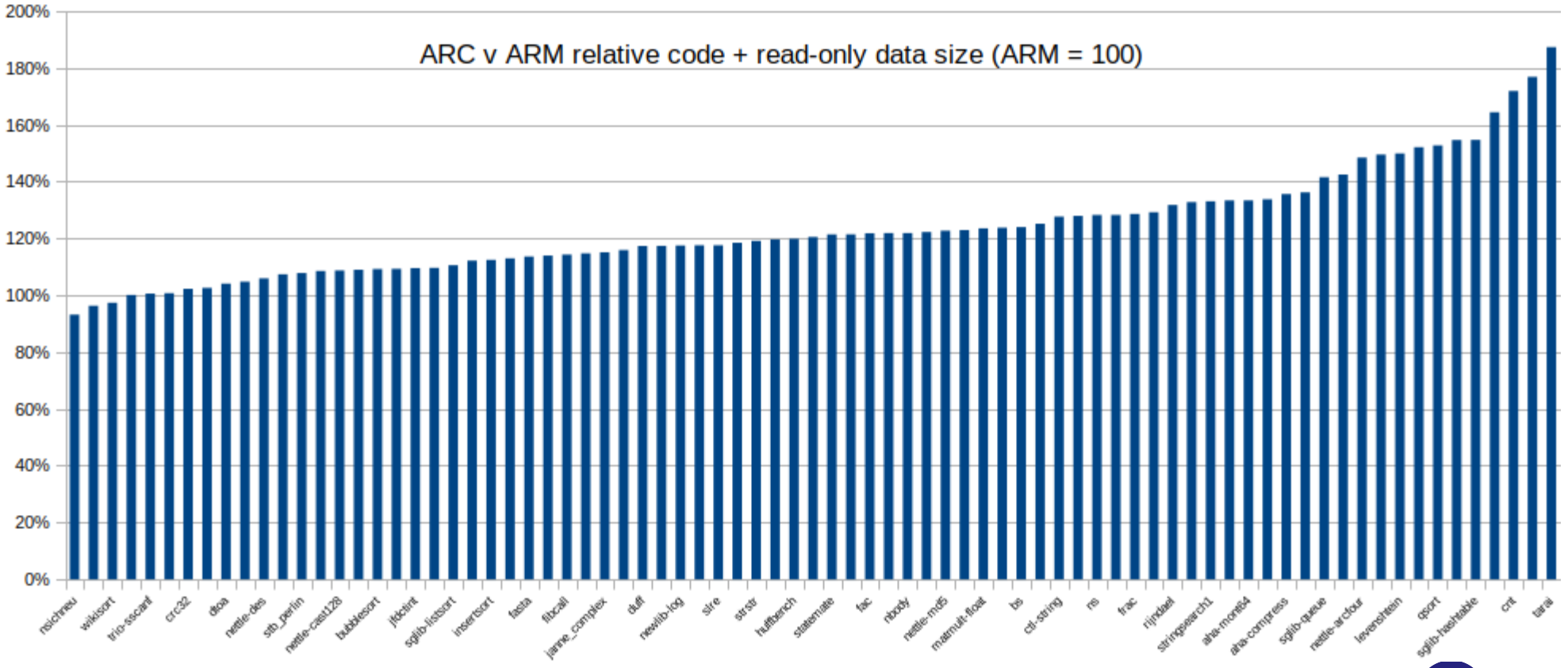
EMBECOSM®

Useful Graph: Arm v RISC-V



EMBECOSM®

Useful Graph: ARC v ARM



EMBECOSM®

So What Did We Learn About GCC?

- New instructions would help
  - add 14-bit constant: 1.1%
  - 48-bit instruction to load 32-bit constant: 1%
- Compiler techniques
  - millicode: 0.33%
  - linker CSE
  - millicode for scaled index load
  - peephole optimization of dead register loads
  - loop rolling

To Do

- More measurements:
  - repeat for LLVM
  - look at DesignWare ARC EM
  - separate out code and read-only data
  - look at initialized writable data
- More compiler analysis
  - focus on the programs that are very different

Resources

- Standard BEEBS
  - http://beebs.eu/
- BEEBS for this talk
  - https://github.com/embecosm/riscv-beebs/tree/grm-size-wip
- Embecosm application note to follow very shortly

EMBECOSM®

Thank You

www.embecosm.com