An Empirical Comparison of the RISC-V and AArch64 Instruction Sets

Daniel Weaver
ra18837@bristol.ac.uk
University of Bristol
Bristol, United Kingdom

Simon McIntosh-Smith
University of Bristol
Bristol, United Kingdom

ABSTRACT

In this work we perform one of the first in-depth, empirical comparisons of the Arm and RISC-V instruction sets. We compare a series of benchmarks compiled with GCC 9.2 and 12.2, targeting the scalar subsets of Arm's Armv-8a and RISC-V's rv64g. We analyse instruction counts, critical paths and windowed critical paths to get an estimate of performance differences between the two instruction sets, determining where each has advantages and disadvantages. The results show the instruction sets are relatively closely matched on the metrics we evaluated for the benchmarks we considered, indicating that neither ISA has a large, inherent advantage over the other, architecturally.

CCS CONCEPTS

• **Hardware** → **Emerging architectures**; *Emerging simulation*; **Emerging languages and compilers**.

KEYWORDS

RISC-V, AArch64, SimEng Simulation, Comparison, Performance

ACM Reference Format:
Daniel Weaver and Simon McIntosh-Smith. 2023. An Empirical Comparison of the RISC-V and AArch64 Instruction Sets. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3624062.3624233

1 INTRODUCTION

Arm's RISC-style instruction set architecture (ISA), Armv8-a, is making inroads into high performance general purpose systems [3], as well as high performance computing applications, for both scientific computing [1] and cloud applications [6]. NVIDIA is set to launch its first Arm-based, high-end CPU, Grace, to be used in the newest generation of supercomputers in the coming months [5, 14]. Various startup companies have indicated they intend to design similar high-end CPUs designed using RISC-V [2, 4].

RISC-V has begun to make progress in the low-level embedded space, with its 10 billionth core shipped last year [7]. This is where Arm have historically dominated, more recently showing that the Arm architecture can transition into high performance use cases in HPC and the cloud.

When designing new hardware, the main trade-offs that are often balanced are Power, Performance and Area (PPA). Both power and area are out of scope for this study, with our main focus being on how performant RISC-V could theoretically be when compared to AArch64. If RISC-V can deliver the performance seen by AArch64 then it should not have any fundamental limiting factors in high performance use cases.

The aim of this work is to compare the differences between AArch64 and RISC-V and how these affect a program's execution time. We direct our focus on the effect differences have on each part of equation 1 in isolation. We do this by analysing path lengths and critical paths for a series of benchmarks aided by the use of The Simulation Engine (SimEng). We choose SimEng as it is a cycle accurate microarchitectural simulator modelling simple atomic cores up to superscalar, out of order cores. It is open source, easy to modify and able to run statically linked real world binaries. This allows us to precisely and easily perform thorough analyse on dynamic instruction traces.

We organise our study as follows: we first precisely define the problem, before discussing the workloads chosen and how these were compiled. We describe our path length methodology, then present and analyse the results. We then estimate the ideal CPI for each ISA through critical path analysis, determining the lowest run times possible for each ISA. Finally we gain more realistic insight into this metric by taking into account physical limits, such as reorder buffer sizes and execution latencies.

1.1 Contributions

This work makes the following contributions

- We have added new RISC-V support to the SimEng microarchitectural simulator.
- We have performed a path length analysis for a series of benchmarks, comparing results between AArch64 and RISC-V, and analysing the reasons behind any differences.
- We provide a critical path analysis to gain theoretical minimum runtimes for a set of benchmarks targeting the two ISAs.
- Finally, we show results from a scaled critical path and windowed critical path analysis, which simulates the effects of structural hazards on the potential runtimes of the two ISAs.

2 PROBLEM DEFINITION

The execution time of a program can be described by the following equation:

This work is licensed under a Creative Commons Attribution International 4.0 License.

SC-W 2023, November 12–17, 2023, Denver, CO, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624233

$$Program\text{-}Execution\text{-}Time = Path\text{-}Length \times CPI \times Time\text{-}per\text{-}Cycle \quad (1)$$

Where Path-Length describes the number of instructions needed to execute the program, CPI (Cycles Per Instruction) describes the average number of cycles needed to execute each instruction, and Time-Per-Cycle is the inverse of the clock rate. For each ISA we want to minimise this equation.

For the same program, factors effecting the path length are the compiler and the ISA. The CPI is effected by the ISA, the specific microarchitecture of the CPU implementing it, and the program being executed. The time per cycle is a function of the microarchitecture, the circuitry implementing it and the underlying physics underpinning the implementation in the silicon manufacturing process. As this is wholly limited by factors other than the ISA, we will keep this constant throughout.

2.1 The Workloads

Five workloads have been chosen for this study, as follows:

- STREAM [11]
  A benchmark for measuring sustained memory bandwidth widely used in industry, this consists of 4 simple kernels applied to elements of arrays of size 10,000,000.
- CloverLeaf Serial [10]
  A high energy physics simulation solving the compressible Euler equations on a 2D Cartesian grid. This is broken down into a series of kernels each of which loops over the entire grid. This is run with default parameter.
- MiniBUDE [12, 15]
  A mini app approximating the behaviour of a molecular docking simulation used for drug discovery.
  Run with the bm1 input at 64 poses for one iteration (`-n 64 -i 1 -deck /bm1`).
- Lattice Boltzmann (LBM)
  A d2q9-bgk Lattice Boltzmann algorithm, developed within the HPC Research Group at the University of Bristol, optimised for serial execution. Run with a grid size of 128x128 for 100 iterations.
- Minisweep [13]
  A radiation transportation mini app reproducing the Denovo Sn radiation transport behaviour used for nuclear reactor neutronics modeling.
  Run with options `-ncell_x 8 -ncell_y 16 -ncell_z 32 -ne 1 -na 32`

These codes have been chosen for their ease of access to source code and amicability to our simulation environment, the main factor being that they do not need full system simulation. Two of these codes are part of SPEC HPC, with a third (LBM) closely matching another SPEC HPC benchmark. STREAM is a well known benchmark with simple kernels that are easy to analyse at the assembly level.

2.2 The Compilers

We chose to use two widely-used compilers, GCC 9.2 and GCC 12.2, and built versions targeting each ISA. When compiling each workload, in addition to the default compiler flags we use `-march=armv8-a+nosimd -mtune=cortex-a55 -static` for AArch64, `-march=rv64g -mtune=sifive-7-series-static` for RISC-V.

Armv8-a by default has SIMD instructions in the form of the NEON extension. RISC-V does have an equivalent to this in the form of its P extension, but this is not yet ratified at the time of writing. The V extension has been recently ratified, but comparing the different vector instruction sets across AArch64 and RISC-V is beyond the scope of this initial comparison; we save this for future work. SIMD instructions would clearly advantage Arm in a performance comparison, so we do not allow the compilers to produce these instructions with `+nosimd`. RISC-V also provides a common set of extensions that are often implemented in processors for general applications, called the G extension. This encapsulates the base (I), atomic (A), multiply/divide (M), single precision floating point (F) and double precision floating point (D) extensions. This is often paired with the compressed (C) extension, but as there is no support for Thumb in Armv8-a, this has been omitted. These choices enable us to make an as "apples to apples" comparison as possible.

There are limited options for the `-mtune` flag when targeting RISC-V. 'Rocket', a series of SiFive processors, which, when digging deeper, use the same cost model[1], 'thead-c906' and an option to tune for 'size' which is "not intended for use by end-users" [2]. We chose to use 'sifive-7-series' as the most powerful of the options, being the only dual issue processor, albeit still only in-order pipelined. To match this we choose to target the Arm Cortex A55; being dual issue, in-order and 8 stages, it closely matches the SiFive core [3]. It should be noted that the cost model for this (and many other Arm CPUs) is far more mature than that of RISC-V; 16 parameters vs ~100[4][5].

Finally, we also compile statically as this is required by the simulation environment we are using. This caused some issues, especially when targeting Armv8-a without SIMD instructions, as SIMD instructions are so inherently tied to the base architecture they are included in all statically linked libraries. This is to be expected in order to be performant, but attempting to build the libraries manually without SIMD is not trivial due to highly optimised functions like `memcpy` using hard coded NEON instructions. The majority of the Arm binaries do not use NEON instructions, but these instructions could not be fully eliminated. The main instances where they are used are to zero out the floating point registers. The RISC-V GNU toolchain must also be built carefully to avoid compressed instructions in archive files, but this was possible.

[1] https://github.com/gcc-mirror/gcc/blob/d073e2d75d9ed492de9a8dc6970e5b69fae20e5a/gcc/config/riscv/riscv.cc#L307
[2] https://gcc.gnu.org/onlinedocs/gcc/RISC-V-Options.html
[3] https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a55
[4] https://github.com/gcc-mirror/gcc/blob/95a2e5328e5aa15724ab8da4aa622a0bfc40c9e5/gcc/config/arm/arm.cc#L2148
[5] https://github.com/gcc-mirror/gcc/blob/95a2e5328e5aa15724ab8da4aa622a0bfc40c9e5/gcc/config/arm/aarch-cost-tables.h#L132

3 PATH LENGTH

3.1 Method

Path length is the number of instructions needed to execute a program. To measure this we must execute each binary and count the number of instructions that were needed to do so. For this we need a simulator that can execute both AArch64 and RISC-V instructions and so we choose The Simulation Engine[6] from the University of Bristol's High Performance Computing group. Initially SimEng did not support execution of RISC-V binaries and so we implemented this execution logic. Once complete, we chose the included emulation core model which executes each instruction atomically to completion in a single cycle, giving a final instruction count.

3.2 Results

Figure 1 shows the results for each benchmark, broken down either by kernel or basic code block. Each set of results is normalised to the result of GCC 9.2 targeting Armv8-a. Raw path lengths can be found in the top row of Table 1.

As seen in the graphs, path lengths for RISC-V and Arm are similar, in most cases within 10% of their compiler version counter part. The largest difference is LBM compiled with GCC 9.2 at 21.7%, and the smallest difference being minisweep compiled with GCC 12.2 at 2.1%. For 6 out of 10 mini-app+compiler pairs, Arm has a shorter path length, with the overall average difference when weighting each benchmark equally being 2.3% longer for RISC-V. This relatively small difference is in large part due to a 16.2% shorter path for RISC-V on miniBUDE.

3.3 STREAM Analysis

To dig deeper into the reason for these differences, we first look at STREAM as the simplest of the benchmarks. Looking at the improvements when moving from GCC 9.2 to GCC 12.2 targeting Arm, we find that for all kernels, the improvements come from changing `sub x1, x0, #2441, lsl #12; subs x1, x1, #1664` to `cmp x0, x20` to set the NZCV register. This results in a single instruction decrease for each kernel, and a 12.5% reduction in path length. In contrast, the main kernels remain the same for both RISC-V binaries. Below is the disassembly of the copy kernel produced by GCC 12.2.

Listing 1: Armv8-a Copy

```
ldr d1, [x22, x0, lsl #3]
str d1, [x19, x0, lsl #3]
add x0, x0, #1
cmp x0, x20
b.ne        0x400abc <main+0x248>
```

Listing 2: rv64g Copy

```
fld fa5,0(a5)
fsd fa5,0(a4)
add a5,a5,8
add a4,a4,8
bne a5,s0,10dec <main+0x240>
```

[6]https://uob-hpc.github.io/SimEng/

Comparing the copy kernels between the AArch64 and RISC-V binaries produced by GCC 12.2, each ISA requires five instructions per element of the array. Both require a single load and store; however, when incrementing the index registers, RISC-V requires two `add` instructions: one for the array being loaded from, and one for the array being stored to. AArch64 on the other hand only requires a single `add` here. This is because of its use of register offset loads and stores, allowing the adding of a base and offset register and performing a load in a single instruction. These instructions also support shifts and zero extensions of the offset register. Consequently, only a single register (X0) is needed to store an offset into the array, as the kernel walks along both arrays at the same rate. Only one register therefore needs to be incremented per cycle. This gives AArch64 a single instruction saving over RISC-V for each iteration of the loop. Immediate offsetting is the only form of load or store in RISC-V and so four instructions is the minimum required to traverse two arrays.

Arm has many variations available for loading and storing data. Simply using a base register or having an immediate offset encoded in the instruction have equivalent single RISC-V instructions. Moving up to register offsets would require two RISC-V instructions for an equivalent sequence, while register offsets with simple shifts and bit extensions would require three RISC-V instructions. Armv8-a also allows for pre and post-indexing of the base register. This increments the base register by the offset amount either before or after the memory operation is performed. Equivalent to a load preceded or succeeded by an add instruction this would require 2 instructions in RISC-V.

In the case of the copy kernel, these instruction would allow for a more optimal solution for AArch64. Both the load and store could be post-indexed with an immediate offset of 8 bytes, thus removing the need for the `add` (and the `shift`), reducing the instructions needed further to only four.

Finally, we focus on the branching logic. For Arm, any conditional branch must be preceded by an instruction setting the state of the NZCV register, either in the form of a `cmp` or a `subs` etc. This may be part of the calculation being performed before the branch or not. If not, this adds an extra single-instruction penalty for conditional branching. This compares to RISC-V, which provides instructions for comparing two registers *and* taking the branch in a single instruction.

RISC-V performs 460,027,962 branches to complete STREAM. This is almost 15% of all instructions executed. If all of these are conditional branches, this is 460 million compare instructions that don't have to be executed compared to the equivalent program running on AArch64. Assuming all other instructions are the same, this slight difference in branching could lead to Arm requiring up to 15% more instructions to execute this workload.

This analysis is consistent across the other three STREAM kernels. AArch64 wins on add and triad due to register indexed loads and stores, and the need to only increment one register instead of three. This gives a 6% difference in path length for the binaries produced by GCC 12.2. It is important to note that while RISC-V had a slightly longer path length, the solution produced is optimal, whereas that produced for Arm is not. This may be a "conscious" decision by the compiler not to use the more complex load and store instructions in AArch64 or a result of a sub-optimal cost model, but

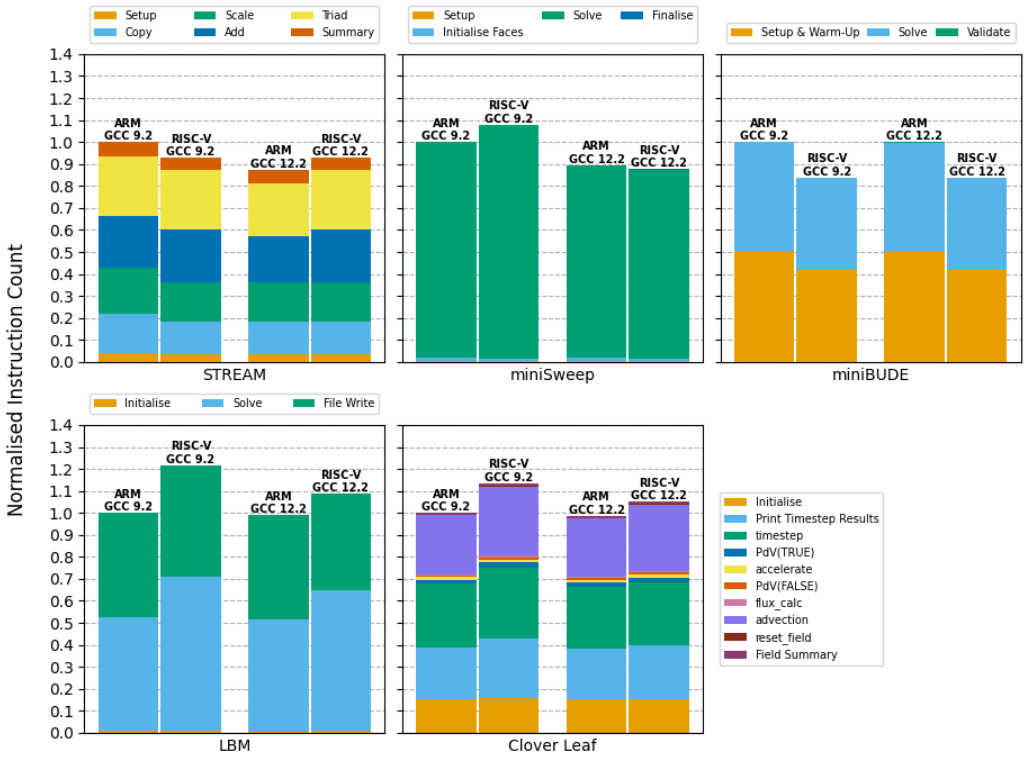

Figure 1: Path lengths for each benchmark broken down by kernel and grouped by compiler

either way, this highlights that with more instructions performing similar functionality, there are more correct solutions which could make it harder for the compiler to choose the best one.

4 IDEAL CPI

Compilers and microarchitectures work in tandem to reduce cycles per instruction (CPI), in order to reduce the overall run time of a program. We will first explore theoretical minimum CPI for each of our workloads, before adjusting our methods to simulate real world structural hazards.

To calculate a theoretical minimum CPI, we first assume that we have an ideal processor which can execute an entire instruction in one cycle, and which can execute any number of instructions in the same cycle, but which also has to obey any true dependencies between instructions. We then use this model to calculate how many cycles it would take to execute our programs. From this, we can calculate the lowest CPI possible using the equation CPI = no. cycles / no. instructions executed. We can then determine minimum theoretical run times using equation 1. This notion of an ideal processor maps onto a modern day Out-of-Order (OoO) superscalar processor that has a single stage, infinite Out-of-Order resources (e.g., ReOrder Buffer (ROB), physical registers, execution units), perfect branch prediction, and single cycle loads and stores.

The only limiting factor for an ideal processor is obeying true or Read-After-Write (RAW) dependencies. The number of cycles it takes to execute a full program is the number of instructions in the longest chain of RAW dependencies. We refer to this chain as the Critical Path (CP). Importantly, this is a property of the program itself and the number of true dependencies that are present, rather than any processor that it runs on. From this we calculate the $CPI = CPlength/pathlength$, the reciprocal of which is the Instruction Level Parallelism (ILP) of the program.

4.1 Method

We again use SimEng to determine the CP. Other tools are available to perform this sort of analysis [9], but these produce full directed acyclic graphs which aren't necessary for our study, instead being designed for in-depth analysis of individual kernels.

Instead, we modify the default SimEng emulation core. Using an array to maintain the critical path length to the value held in each register, and a map to keep track of path lengths for each memory address used, we can then execute the program through the core as normal.

For each instruction executed we determine the source and destination registers and the source and destination memory addresses. This data is easily gathered as SimEng disassembles and decodes each instruction for simulation. We index the register array and memory map with the sources for this instruction. This gives the current longest chain of dependencies for each source. We take the longest of these dependencies, add one for the instruction currently being executed, and write this value to the array and map, indexed

Table 1: Critical Paths and ILP per Benchmark

	STREAM				CloverLeaf			
	GCC 9.2		GCC 12.2		GCC 9.2		GCC 12.2	
	AArch64	RISC-V	AArch64	RISC-V	AArch64	RISC-V	AArch64	RISC-V
Path Length	3,350,107,615	3,110,150,358	2,930,114,073	3,110,139,144	12,832,452	14,553,390	12,647,061	13,481,498
CP	10,000,234	10,005,341	10,000,234	10,004,815	46,933	191,538	46,658	228,036
ILP	335	311	293	311	273	76	271	59
2GHz Run time (ms)	5.00	5.00	5.00	5.00	0.0235	0.0958	0.0233	0.114

	LBM				miniBUDE			
	GCC 9.2		GCC 12.2		GCC 9.2		GCC 12.2	
	AArch64	RISC-V	AArch64	RISC-V	AArch64	RISC-V	AArch64	RISC-V
Path Length	380,391,346	463,305,683	376,329,390	412,979,829	137,280,541	115,064,988	137,183,536	114,897,049
CP	10,910,427	5,196,321	4,660,144	4,873,467	196,357	197,285	196,331	196,722
ILP	35	89	81	85	699	583	699	584
2GHz Run time (ms)	5.46	2.60	2.33	2.44	0.0982	0.0986	0.0982	0.0984

	minisweep			
	GCC 9.2		GCC 12.2	
	AArch64	RISC-V	AArch64	RISC-V
Path Length	2,162,866,809	2,332,356,452	1,934,709,957	1,894,737,614
CP	263,120	263,327	280,567	272,444
ILP	8,220	8,857	6,896	6,955
2GHz Run time (ms)	0.132	0.132	0.140	0.136

with the destination registers and memory addresses. The zero register for each ISA always reads zero. We keep track of the globally longest CP and update this for each instruction.

This method is slightly naive. It does break CPs when sources of an instruction are the zero register; however, it is hard to determine whether other ways of zeroing out a register are just the result of a calculation or an intentional clearing of data. Methods such as bitwise XOR-ing a register with itself are not detected as breaking the CP. Similarly, repeated mapping and unmapping memory addresses does not break chains through memory. This behaviour does not occur when running any of our benchmarks.

4.2 Results

Table 1 shows the CPs for each benchmark. The ILP has been calculated and then a runtime estimated assuming a 2GHz clockspeed, similar to that of modern day application level processors. [7] For benchmarks STREAM, miniBUDE and minisweep, estimated runtimes for both ISAs are very similar; within 1% for the first two and 2.9% for the third. MiniBUDE had one of the largest differences in path length between ISAs, while reporting almost exactly the same critical path length. But, as runtime is purely a function of the CP, these are also almost the same showing that Arm has more instruction level parallelism theoretically available for this code.

For LBM, GCC 9.2 Arm has higher runtimes than the rest for which we currently don't have the reason. CloverLeaf RISC-V has path lengths ~4X that of Arm leading to far larger runtimes the reasons for which we also haven't determined.

ILP here is a metric of the average number of instructions we would have to execute on every cycle to achieve this optimal run time. Values in the hundreds are clearly not realistic today, as this would also require hundreds of execution units, not to mention all the supporting hardware throughout the rest of the pipeline. To begin to address this, and to provide more realistic data, we first look at the effect of differing execution times per instruction on the critical path.

5 SCALED CRITICAL PATH

To allow for quicker clock rates in real processors, more complex instructions are calculated over a series of cycles, while simpler instructions may take as few as a single cycle. This will clearly effect the length of the critical path, but also possibly its composition.

5.1 Method

SimEng provides accurate models for Marvell's ThunderX2 (TX2), Fujitsu's A64FX and Apple's M1 Firestorm cores, all based on AArch64, we defined a RISC-V core model based off of the TX2 microarchitecture and latencies. These are defined within yaml files with, among others, fields to define the latencies for a group of instructions. Models can be found under the `/configs` directory of the SimEng Git repository.[6] Within the simulation, upon instruction decode each instruction is categorised and given the execution

[7] https://en.wikichip.org/wiki/cavium/thunderx2/cn9980

Table 2: Scaled Critical Paths and ILP per Benchmark

	STREAM				CloverLeaf			
	GCC 9.2		GCC 12.2		GCC 9.2		GCC 12.2	
	AArch64	RISC-V	AArch64	RISC-V	AArch64	RISC-V	AArch64	RISC-V
Scaled CP	60,000,545	60,005,845	60,000,545	60,005,845	94,983	191,538	81,925	244,103
ILP	56	52	49	52	135	76	154	55
2GHz Run time (ms)	30.0	30.0	30.0	30.0	0.0475	0.0958	0.0410	0.122

	LBM				miniBUDE			
	GCC 9.2		GCC 12.2		GCC 9.2		GCC 12.2	
	AArch64	RISC-V	AArch64	RISC-V	AArch64	RISC-V	AArch64	RISC-V
CP	42,344,992	5,888,686	4,660,233	5,565,925	685,839	685,842	685,680	685,291
ILP	9.0	79	81	74	168	168	168	168
2GHz Run time (ms)	21.2	2.94	2.33	2.78	0.343	0.343	0.343	0.343

	minisweep			
	GCC 9.2		GCC 12.2	
	AArch64	RISC-V	AArch64	RISC-V
CP	1,577,198	1,586,189	1,592,550	1,577,099
ILP	1,371	1,470	1,215	1,201
2GHz Run time (ms)	0.790	0.793	0.796	0.789

latency defined within the yaml file. As our RISC-V model already approximates the latencies provided for the TX2 model, we will use these for this experiment. The TX2 is a good model to use, as it is a classic, 4-way superscalar, OoO RISC microarchitecture, with "typical" latencies for most of its instructions. In this way, TX2 makes a good canonical model on which to base these experiments.

To calculate the critical path accounting for instruction latencies, we first take the algorithm above, and instead of adding one to the current longest CP of the sources, we add the execution latency for the instruction currently being executed. We do not scale for loads or stores as we assume store forwarding in most cases.

5.2 Results

Results can be found in Table 2. For CloverLeaf, runtimes for Arm almost double, while those for RISC-V stay the same. This shows that while critical paths were shorter for Arm here, they were more computationally dense and this is penalised by taking more cycles for more complex instruction. RISC-V path lengths remain almost unchanged, but still between 2X and 3X longer than Arm.

Both ISAs are still on par for LBM, miniBUDE, minisweep and STREAM. LBM CPs increase very slightly for RISC-V, remain for GCC 12.2 targeting Arm and 4X for GCC 9.2 for which the reason is currently unclear. MiniBUDE CPs scale evenly by 3.5X, minisweep by 6X and STREAM by 6X. In these cases where scaling is the same between ISAs, instructions along the critical path have an almost 1-to-1 correspondence owing to the same amount of scaling.

These results are interesting theoretically, but they aren't representative for real microarchitectures, as the method still assumes an infinite number of resources. To address this, we perform a windowed critical path analysis, simulating a finite-sized ROB.

6 WINDOWED CRITICAL PATH

A ROB determines the maximum number of instructions that can be in flight at any one time and the pool of instructions available for the execution units to draw upon. To naively simulate this we perform a windowed critical path analysis.

6.1 Method

Sliding a window of differing sizes over the full execution path, we determine the critical path for the set of instructions in the current window, moving the window 50% of its size further along the path once this is done. E.g. for a window size of four, we first look at the CP of the first four instructions, then instructions 2-6, then 4-8. The idea of the window maps onto an OoO processor with perfect branch prediction, that is able to fetch a window size of instructions in one cycle, filling the ROB. The ROB is the size of the window and there are infinite physical registers. Sliding this window by fewer instructions than the window size can either be thought of as having a limited commit stage wide enough for the amount of instructions that the window is slid by, or as a proxy for limited execution units. Due to time constraints we do not adjust this value. We also do not account for instruction latency.

We choose window sizes of 4, 16, 64, 200, 500, 1000 and 2000. This balances runtime for our simulations, whilst also giving a wide range of hypothetical ROB sizes. These represent cases where a ROB may be empty and just filling from a limited superscalar front end, up to the largest modern day ROBs [8] at ~630 entries and beyond. For simplicity, for this part of the study we only look at binaries produced by GCC 12.2.

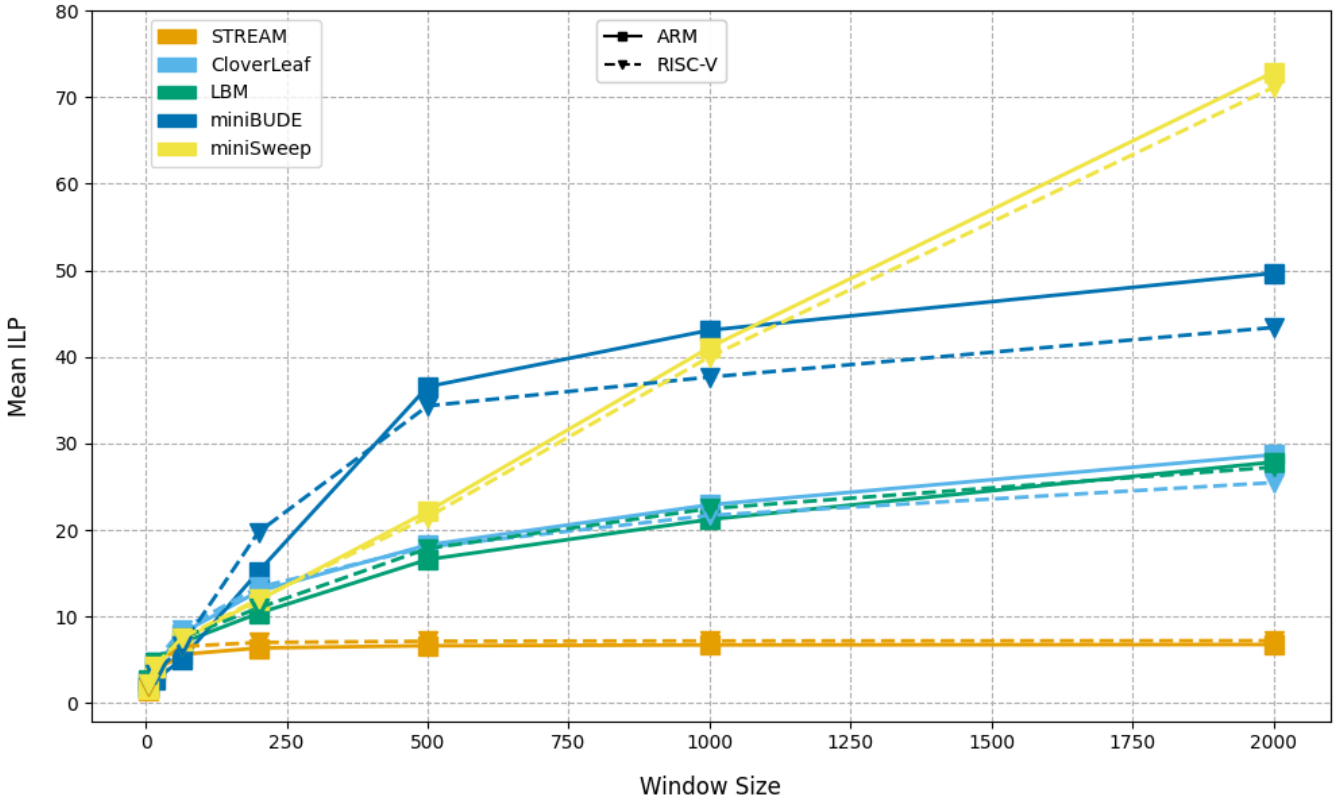

Figure 2: Mean ILP per window

6.2 Results

From this analysis we get a large range of critical path lengths for a specific window size. Figure 2 presents the mean ILP plotted against window size.

Due to the regular nature of STREAM, many of the windows produced CP lengths of the same size, with no CP lengths ± 1 instruction. All other benchmarks had much smoother distributions of CP lengths.

In all cases, the ISAs track each other closely. The largest difference is for CloverLeaf at a window size of 2000, where RISC-V has 12% less ILP available. The only case where RISC-V has more ILP at large window sizes is STREAM with a 5.8% advantage. In every case however, at lower window sizes (500 or less), RISC-V has more ILP available with AArch64 overtaking at higher window sizes. This shows that local dependent instructions are more distantly spread for RISC-V which could allow for increased throughput in OoO processors. This effect appears to be small though.

7 CONCLUSIONS

Our results show that the RISC-V ISA is not disadvantaged compared to the Arm AArch64 ISA in terms of potential performance for codes produced by today's compilers for HPC style workloads. With instruction counts largely within 10% of each other, and equivalent parallelism available, in most cases this leads to estimated equivalent run times for these architectures.

Deeper analysis of STREAM shows that, with Arm's more powerful load and store instructions, path lengths have the potential to be much shorter than those presented. But with so many options available, the optimal solution is harder for compilers to find. In addition, with the inclusion of comparison instructions, AArch64 binaries require additional instructions when conditionally branching compared to RISC-V, potentially leading to up to 15% longer paths with all other instructions equivalent.

8 FUTURE WORK

Clearly, more than just the critical path matters in the real world. With finite sized ROBs and fetch units a processor only has limited insight into the program it is executing. Any chain of dependencies could be part of the CP and with limited execution units one can't be prioritised over another.

SimEng provides the capability for simulating OoO superscalar microarchitectures which are easily modifiable. We plan to perform similar analysis through this simulation, using real-world sizes for OoO resources, while also extrapolating to hypothetical microarchitectural designs of the future. This should give accurate estimations

of runtimes for both ISAs for processors of today and the near future.

ACKNOWLEDGMENTS

To Rahat Muneeb for help producing graph. To ExCALIBUR H&ES RISC-V testbed for access to RISC-V compilers, and to members of Bristol HPC for proofreading and continual support. This work was supported by PhD funding from EPSRC and Huawei.

REFERENCES

[1] 2020. GW4 Supercomputer Isambard. https://gw4.ac.uk/case-studies/gw4-supercomputer-isambard/
[2] 2022. https://www.prnewswire.com/news-releases/ventana-introduces-veyron-worlds-first-data-center-class-risc-v-cpu-product-family-301700985.html
[3] 2022. Apple unveils M1 Ultra, the world's most powerful chip for a personal computer. https://www.apple.com/newsroom/2022/03/apple-unveils-m1-ultra-the-worlds-most-powerful-chip-for-a-personal-computer/
[4] 2023. https://riscv.org/blog/2023/01/spacemit-makes-important-breakthroughs-in-risc-v-high-performance-cores-spacemit/
[5] Paul Alcorn. 2022. Nvidia's Grace CPU Superchip to power two supercomputers, up to ten "ai exaflops". https://www.tomshardware.com/news/nvidias-grace-cpu-superchip-to-power-two-supercomputers-up-to-ten-ai-exaflops
[6] Donald J. Daly. 2019. Graviton2. https://aws.amazon.com/ec2/graviton/
[7] Nick Flaherty. 2022. Europe steps up as RISC-V ships 10bn cores. https://www.eenewseurope.com/en/europe-steps-up-as-risc-v-ships-10bn-cores/
[8] Andrei Frumusanu. 2020. Apple announces the Apple Silicon M1: Ditching x86 - what to expect, based on A14. https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2
[9] Jan Laukemann, Julian Hammer, Georg Hager, and Gerhard Wellein. 2019. Automatic throughput and critical path analysis of x86 and arm assembly kernels. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 1–6.
[10] Andrew Mallinson. 2013. Cloverleaf: Preparing hydrodynamics codes for exascale. (2013).
[11] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.
[12] Simon McIntosh-Smith, James Price, Richard B Sessions, and Amaurys A Ibarra. 2015. High performance in silico virtual drug screening on many-core processors. *The international journal of high performance computing applications* 29, 2 (2015), 119–134.
[13] OE Bronson Messer, Ed D'Azevedo, Judy Hill, Wayne Joubert, Mark Berrill, and Christopher Zimmer. 2018. MiniApps derived from production HPC applications using multiple programing models. *The International Journal of High Performance Computing Applications* 32, 4 (2018), 582–593. https://doi.org/10.1177/1094342016668241 arXiv:https://doi.org/10.1177/1094342016668241
[14] Oliver Peckham. 2023. Nvidia, HPE announce Superchip-powered "isambard 3" supercomputer. https://www.hpcwire.com/2023/05/21/nvidia-hpe-announce-superchip-powered-isambard-3-supercomputer/
[15] Andrei Poenaru, Wei-Chen Lin, and Simon McIntosh-Smith. 2021. A performance analysis of modern parallel programming models using a compute-bound application. In *International Conference on High Performance Computing*. Springer, 332–350.

A ARTIFACT APPENDIX

A.1 Abstract

SimEng is a C++ based microarchitectural simulator with the aims to be fast, easily modifiable and accurate. This is used as a base which we modified to perform our experiments. Scripts that download and install SimEng then subsequently reproduce all of our results are openly available.

A.2 Artifact check-list (meta-information)

- **Program:** SimEng, STREAM, miniBUDE, Lattice Boltzmann (private, serially optimised version of https://github.com/UoB-HPC/advanced-hpc-lbm), CloverLeaf_serial, Minisweep. Minimal sources and binaries included
- **Compilation:** C++17 compatible compiler, GCC 9.2 and 12.2 able to target AArch64+nosimd and rv64g
- **Binary:** Binaries for all benchmarks included
- **Run-time environment:** Tested on Ubuntu 22.04 with git, cmake, Python3 with NumPy and Matplotlib
- **Hardware:** Results reproduced on x86_64. Benchmarks compiled with AArch64 hardware (ThunderX2)
- **Metrics:** Instruction counts, critical paths, windowed critical paths as output from SimEng
- **Output:** Console output, pdfs of graphs, text files with data used in tables expected to be as in the paper
- **Experiments:** Scripts with small variation possible in instruction counts based on host OS
- **How much disk space required (approximately)?:** 0.5GB
- **How much time is needed to prepare workflow (approximately)?:** 5 minutes
- **How much time is needed to complete experiments (approximately)?:** 9 hours on a powerful laptop
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** Apache-2.0, MIT, BSD-3-Clause, BSD-2-Clause
- **Workflow framework used?:** Bash scripts and Python scripts
- **Archived:** https://github.com/UoB-HPC/Arm-RISCV-Empirical-Comparison-Artifact

A.3 Description

A.3.1 How to access. Vanilla SimEng can be obtained through GitHub at https://github.com/UoB-HPC/SimEng with documentation at https://uob-hpc.github.io/SimEng/. Scripts, binaries and benchmark sources recreating the results of this paper can be found at https://github.com/UoB-HPC/Arm-RISCV-Empirical-Comparison-Artifact

A.3.2 Hardware dependencies. No specific hardware is needed to run SimEng (although it is only supported on MacOS and Linux) or compile the benchmarks from source. For ease however, it is recommended to compile the benchmarks on hardware implementing the ISA that is being targeted. For example, we used the XCI nodes of Isambard2 to compile AArch64 binaries https://gw4-isambard.github.io/docs/user-guide/XCI.html. The scripts recreating our results are single threaded and so a single core of a modern laptop will be sufficient to reproduce them.

A.3.3 Software dependencies. To build SimEng you will need git, cmake and some C++17 compatible compiler. To run the build script and reproduce the graphs in this paper you will need the above as well as Python3 with the NumPy and Matplotlib packages. To build the benchmarks GCC 9.2 and 12.2 are needed targeting AArch64 and rv64g instruction sets. The scripts reproducing the results have been successfully tested on Ubuntu 22.04 with these dependencies.

A.4 Installation

Once the above software dependencies have been installed, download the artifact from https://github.com/UoB-HPC/Arm-RISCV-Empirical-Comparison-Artifact. To replicate the results in this paper run `source buildAndRun.sh`. Note, when rerunning this script previous output and results will be deleted.

A.5 Experiment workflow

The buildAndRun script performs many actions. First generating two directories: `output` for raw SimEng output per set of benchmark binaries, and `results` for processed information gained from the raw output as well as graphs. SimEng is then git cloned from GitHub. For each experiment (path length count per kernel, critical path length, scaled critical path length, and windowed critical path length) the same set of operations are performed: git checkout the relevant commit, build SimEng, run all relevant (pre-compiled) binaries and save the output under the `output` directory, and finally perform some analysis with a python script. Results from these scripts are stored in the `results` directory.

A.6 Evaluation and expected results

The `results` directory will contain reproductions of the two graphs as well as four text files. `kernelCounts.txt` contains the relevant raw output from SimEng, Python dictionaries containing the cumulative instruction count per section of the source code, and at the bottom, normalised counts per section used as input to `grouped_stacked_bar.py`. `basicCPResult.txt` and `scaledCPResult.txt` contain critical path data taken from raw SimEng output as well as ILP per benchmark. `windowAverages.txt` contains comma separated lists per benchmark. Each element is the mean CP length per window size in ascending order. This is then used to produce `lineGraph.pdf`. It is expected that results will vary slightly depending on host system but the trends should remain.

A.7 Experiment customization

SimEng is open source and runs real statically linked AArch64 and RISC-V binaries. It should be easy to compile other benchmarks targeting the relevant architectures and run them through SimEng following the documentation https://uob-hpc.github.io/SimEng/. Occasionally, an instruction won't be implemented, in which case an issue should be submitted on the GitHub page.