# Static and Dynamic ICOUNT using GNU Toolchain and QEMU

This section presents statistics and some analysis on the total number of static and dynamic instructions executed for three different ISAs (MIPS, ARM, and RISCV) for both 32bit and 64bit architectures. Results are shown for 11 different applications from a standard embedded system’s benchmark suite called MiBench. These applications target a variety of activities suitable for embedded systems. Results and micro-architecture statistics using gem5 cycle-accurate simulator (e.g. total number of cycles, cache misses, etc.) will be shown in the next section.

For collecting static and dynamic icount, we use well-known QuickEMUlator (QEMU) emulator to run binaries for different ISAs on an x86 host machine. QEMU is a free and open-source emulator that performs hardware virtualization and emulates the machine's processor through dynamic binary translation and provides a set of different hardware and device models for the machine, enabling it to run a variety of guest operating systems. It also can be used with KVM to run virtual machines at near-native speed (by taking advantage of hardware extensions such as IntelVT). QEMU can also do emulation for user-level processes, allowing applications compiled for one architecture to run on another. We use this user-level emulation feature to run different binaries (for a given application) on an x86 machine.

For the dynamic icount, QEMU is modified to count number of executed instructions (target instruction) and prints the total number of instructions at the end of the program. For static icount, QEMU is modified to count number of unique (target) PCs during the execution of each application. In other words, static icount in this document, shows the number instructions that are executed at least once and not just all the instructions that exist in the binary (we found that these two number could be quite different for some applications since gcc toolchain might include source codes for all routines in an included library, while some of these routines may not be used at all).

Figure 1 and 2 show the results for dynamic and static ICOUNT respectively for 11 Mibench applications using QEMU and GNU toolchain to cross-compile each of applications to MIPS, ARM, and RICV.

Figure 1. Total number of dynamic executed instructions for 11 applications from MiBench benchmark suite for 3 different architectures.

Figure 2. Total number of dynamic executed instructions for 11 applications from MiBench benchmark suite for 3 different architectures.

Followings provides some detail about each application and also presents some explanations for why dynamic and/or static icounts are different for different architectures.

**Basicmath**: This application performs simple mathematical calculations that often don’t have dedicated hardware support in embedded processors. For example, cubic function solving, integer square root and angle conversions from degrees to radians are all necessary calculations for calculating road speed or other vector values. The input data is a fixed set of constants.

Dynamic icount for 32bit vs. 64bit versions are very different to that of for MIPS and ARM ISAs (the 32bit versions have significantly lower number of dynamic icount). The main reason for this difference comes from the way *solveCubic* function is being executed where a set of *long double* variables are multiplied/divided to each other. For 64bits, MIPS and ARM compilers used a *software floating point library (e.g. multf3, divf3)* to implement and emulate these *long double* multiplications (although *–hardfloat* flag is used during compilations), whereas for 32bit, compilers ignore the long double format and treat it as 64-bit double variables and hence use hardware floating point units which substantially reduce the number of dynamic instructions (for 32bit). To further investigate this, we used *-softfloat* flagfor both 32bit and 64bit ISAs and confirmed that the dynamic icounts are almost similar for both modes when software libraries are only used. We also changed the *long double* to *double* and confirmed that both architectures use hardware floating point units in this case.

For RISCV, since the Q (quad-precision ISA extension) is still not a part of released *gcc* compiler, for both 32 and 64bit RISCV ISAs the results are like that of for 64bit ARM/MIPS (i.e. using soft-float libraries). Moreover, we observed a significantly larger number of dynamic icount for 32bit RISCV. Looking deeper showed that RISCV-32 requires lots of extra *load/store* register instructions and loops/counters to successfully implement two long double multiplication/division which caused almost double size of static code size for these functions (i.e. multf3, divf3, etc.) in RISCV32 and significantly larger number of dynamic instructions.

**Bitcount**: This application tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers. It does this using five methods including an optimized 1-bit per loop counter, recursive bit count by nibbles, non-recursive bit count by nibbles using a table look-up, non-recursive bit count by bytes using a table look-up and shift and count bits. The input data is an array of integers with equal numbers of 1’s and 0’s.

The only outlier in this benchmark is the dynamic icount for ARM32. Looking deeper into the code, we found that the main reason for inefficiency in the ARM32 code is due to the way a very hot loop is being implemented in ARM32. Following shows this loop and its assembly code for ARM32 and ARM64:

Source code:



RISCV32



ARM32:



ARM64:



The main difference between ARM64 and 32 is the “ccmp” instruction. For ARM32, 7 instructions needed to perform same functionality as ccmp. Given that this loop is executed more than 30 million times, 6 extra instructions cause a significant overhead (more than 180 million instructions) on ARM32. Moreover, results for static icount show that ARM64 overall have lower number of instructions which in turn causes a lower number of dynamic icount (compared to RISCV and MIPS).

**Qsort**: The qsort test sorts a large array of strings into ascending order using the well-known *quick sort* algorithm. Sorting of information is important for systems so that priorities can be made, output can be better interpreted, data can be organized, and the overall run-time of programs reduced. The small data set is a list of words; the large data set is a set of three-tuples representing points of data.

There are two interesting observations for this benchmark. First, both ARM and MIPS have higher dynamic icount than RISCV. Second, ARM32 has much higher dynamic icount than ARM64.

For ARM32 vs. ARM64 there are two sources for difference: first, a hot loop in *msort\_with\_temp* function (part of glibc library for quicksort) executes exactly twice for ARM32 since the loop counter depends on the size of the variables (i.e. counter+= sizeof(x)) where for 32bit architecture it is 4 (bytes) and for 64 it is 8. The second difference is in *vfscanf* function where some same functionality requires a few more instructions for ARM32.

Finally, for RISCV, vfscanf executes lower number of instructions to that of in ARM and MIPS which caused lower number of total dynamic icount for RISCV since this one of the dominant functions in qsort. Looking into the source for vfscanf, RISCV vfscanf is quite different than existing functions for ARM or x86 (unlike most of the gnu libraries where RISCV uses (almost) same routines and source codes).

**Susan**: Susan is an image recognition package. It was developed for recognizing corners and edges in Magnetic Resonance Images of the brain. It is typical of a real-world program that would be employed for a vision-based quality assurance application. It can smooth an image and has adjustments for threshold, brightness, and spatial control. The small input data is a black and white image of a rectangle while the large input data is a complex picture.

In Susan, ARM32 and ARM64 have the lowest dynamic than MIPS and RISCV. Following shows a same piece of code (hottest region of SUSAN) for MIPS64, ARM64, and RISCV64



MIPS:



ARM:



RISCV:



ARM compiler manages to merge two counters for \*dpt and \*ipt into one which saves one addition (+1). Furthermore, using sub, ARM further saves another addition for loading \*(cpt-brightness) (another +1). Finally, using multiplyAdd (MADD) instruction, ARM could execute “total += tmp \* brightness;” in only one instructions, whereas RISCV and MIPS need two. In total, ARM saves two instructions out of 12 (i.e. 17% reduction).

**Dijkstra:** The Dijkstra benchmark constructs a large graph in an adjacency matrix representation and then calculates the shortest path between every pair of nodes using repeated applications of Dijkstra’s algorithm. Dijkstra’s algorithm is a well-known solution to the shortest path problem and completes in O() time.

In this application, ARM32 and 64 have much higher dynamic icount than the other two ISAs. The reason for this difference is that each conditional branch in ARM requires two instructions (cmp,bxx), while for RISCV and MIPS, a cmp is not needed. In the hot region shown below for Dijkstra, there are four of such cmp,br groups which incurs 4 extra instructions in ARM (i.e. 22% overhead).

ARM:



RISCV:



**Blowfish:** Blowfish is a symmetric block cipher with a variable length key. Since its key length can range from 32 to 448 bits, it is ideal for domestic and exportable encryption. The input data sets are a large and small ASCII text file of an article found online.

In Blowfish, RISCV has significantly lower number of executed instructions. The main reason for this difference comes from the way *IO*\_*feof* is implemented in RISCV compared to other architectures. Following shows the glibc source code for RISCV vs. ARM. Note the difference between these two codes that shown in red. For RISCV, the code first checks whether a lock is needed or not and then calls \_IO\_flockfile. While in ARM and MIPS it always calls flockfile. Interestingly, for Blowfish, lockfile is not required which causes a significant saving in number of instructions for RISCV.

ARM and MIPS:

**

RISCV:



**rijndael**: Rijndael was selected as the National Institute of Standards and Technologies Advanced Encryption Standard (AES). It is a block cipher with the option of 128-, 192-, and 256-bit keys and blocks. The input data sets are the same as the ones used by blowfish.

In rijnadael, RISCV has significantly larger static code size and that also caused a significantly larger dynamic icount since most of the code is being executed for almost equal number of iterations. Looking into the RISCV code there are a couple of interesting observations: first, for some of the complex ARM instructions, RISCV needs to execute multiple instructions (e.g. UBX in ARM translates into multiple mov and srli instructions, same for add shift combo in ARM, etc.). Second, RISCV compiler tends to use lots of sd/ld instructions (load and store to register) while ARM compiler tends to use more register to avoid issuing extra load/store commands. This is due to the unique ability in ARM ISA, where a load can be combined with lsr/lsl and/or multiple registers can be used for calculating the address/offset for a load instruction. However, in RISCV, only one register can be used to calculate the offset, thus extra activity is needed prior to a load instruction.

**Sha:** SHA is the secure hash algorithm that produces a 160-bit message digest for a given input. It is often used in the secure exchange of cryptographic keys and for generating digital signatures. It is also used in the well-known MD4 and MD5 hashing functions. The input data sets are the same as the ones used by blowfish.

This application spends most of its time in a function called *sha\_transform.* Using some optimizations, ARM compiler manages to significantly reduce number of iterations for some of the loops in this function which leads to large dynamic icount savings for ARM, i.e. ARM compiler can somehow unroll the loops quite more efficiently than that of in RISCV or MIPS. Following shows a snippet of code from *sha\_transform* and shows the compiled code for ARM64 and RISCV64.



ARM:

0000000000400610 <sha\_transform>:

400610: d10a83ff sub sp, sp, #0x2a0

400614: 9100e001 add x1, x0, #0x38

400618: d2801002 mov x2, #0x80 // #128

40061c: a9007bfd stp x29, x30, [sp]

400620: 910003fd mov x29, sp

400624: f9000bf3 str x19, [sp, #16]

400628: aa0003f3 mov x19, x0

40062c: 910083a0 add x0, x29, #0x20

400630: 94005748 bl 416350 <memcpy>

400634: 3dc00fb5 ldr q21, [x29, #48]

400638: 528000e0 mov w0, #0x7 // #7

40063c: 3dc013b4 ldr q20, [x29, #64]

400640: 910283a1 add x1, x29, #0xa0

400644: 3dc017b3 ldr q19, [x29, #80]

400648: 3dc01bb2 ldr q18, [x29, #96]

40064c: 3dc01fb1 ldr q17, [x29, #112]

400650: 3dc023b0 ldr q16, [x29, #128]

400654: 3dc027a7 ldr q7, [x29, #144]

400658: 14000002 b 400660 <sha\_transform+0x50>

40065c: aa0303e1 mov x1, x3

400660: 3cde8026 ldur q6, [x1, #-24]

400664: 11001c00 add w0, w0, #0x7

400668: 3cd80025 ldur q5, [x1, #-128]

40066c: 91016024 add x4, x1, #0x58

400670: 6e321cc6 eor v6.16b, v6.16b, v18.16b

400674: 9100c025 add x5, x1, #0x30

400678: 3cd90024 ldur q4, [x1, #-112]

40067c: d1004026 sub x6, x1, #0x10

400680: 3cda0023 ldur q3, [x1, #-96]

400684: 9101c023 add x3, x1, #0x70

400688: 6e351cc6 eor v6.16b, v6.16b, v21.16b

40068c: 71008c1f cmp w0, #0x23

400690: 3cdb0022 ldur q2, [x1, #-80]

400694: 3cdc0021 ldur q1, [x1, #-64]

400698: 6e251cd5 eor v21.16b, v6.16b, v5.16b

40069c: 3cdd0020 ldur q0, [x1, #-48]

4006a0: 3cde0036 ldur q22, [x1, #-32]

4006a4: 3d800035 str q21, [x1]

4006a8: 3cdf8025 ldur q5, [x1, #-8]

4006ac: 6e311ca5 eor v5.16b, v5.16b, v17.16b

4006b0: 6e341ca5 eor v5.16b, v5.16b, v20.16b

4006b4: 6e241cb4 eor v20.16b, v5.16b, v4.16b

4006b8: 3d800434 str q20, [x1, #16]

4006bc: 3cc08024 ldur q4, [x1, #8]

4006c0: 6e301c84 eor v4.16b, v4.16b, v16.16b

4006c4: 6e331c84 eor v4.16b, v4.16b, v19.16b

4006c8: 6e231c93 eor v19.16b, v4.16b, v3.16b

4006cc: 3d800833 str q19, [x1, #32]

4006d0: 3cc18023 ldur q3, [x1, #24]

4006d4: 6e271c63 eor v3.16b, v3.16b, v7.16b

4006d8: 6e321c63 eor v3.16b, v3.16b, v18.16b

4006dc: 6e221c72 eor v18.16b, v3.16b, v2.16b

4006e0: 3d800c32 str q18, [x1, #48]

4006e4: 3cc28022 ldur q2, [x1, #40]

4006e8: 6e311c42 eor v2.16b, v2.16b, v17.16b

4006ec: 6e211c42 eor v2.16b, v2.16b, v1.16b

4006f0: 6e351c51 eor v17.16b, v2.16b, v21.16b

4006f4: 3d801031 str q17, [x1, #64]

4006f8: 3cc38021 ldur q1, [x1, #56]

4006fc: 6e301c21 eor v1.16b, v1.16b, v16.16b

400700: 6e201c21 eor v1.16b, v1.16b, v0.16b

400704: 6e341c30 eor v16.16b, v1.16b, v20.16b

400708: 3d801430 str q16, [x1, #80]

40070c: 3cc48020 ldur q0, [x1, #72]

400710: 6e271c00 eor v0.16b, v0.16b, v7.16b

400714: 6e361c00 eor v0.16b, v0.16b, v22.16b

400718: 6e331c07 eor v7.16b, v0.16b, v19.16b

40071c: 3d801827 str q7, [x1, #96]

400720: 54fff9e1 b.ne 40065c <sha\_transform+0x4c> // b.any

RISCV:

000000000001033a <sha\_transform>:

1033a: d5010113 addi sp,sp,-688

1033e: 03850593 addi a1,a0,56

10342: 2a813023 sd s0,672(sp)

10346: 08000613 li a2,128

1034a: 842a mv s0,a0

1034c: 850a mv a0,sp

1034e: 29313423 sd s3,648(sp)

10352: 2a113423 sd ra,680(sp)

10356: 28913c23 sd s1,664(sp)

1035a: 29213823 sd s2,656(sp)

1035e: 4b6100ef jal ra,20814 <\_\_memcpy>

10362: 78a6 ld a7,104(sp)

10364: 7846 ld a6,112(sp)

10366: 7566 ld a0,120(sp)

10368: 6642 ld a2,16(sp)

1036a: 66e2 ld a3,24(sp)

1036c: 7702 ld a4,32(sp)

1036e: 72a2 ld t0,40(sp)

10370: 7fc2 ld t6,48(sp)

10372: 7f62 ld t5,56(sp)

10374: 878a mv a5,sp

10376: 45c1 li a1,16

10378: 04f00993 li s3,79

1037c: 0407be83 ld t4,64(a5)

10380: 0487be03 ld t3,72(a5)

10384: 0507b303 ld t1,80(a5)

10388: 0007b903 ld s2,0(a5)

1038c: 6784 ld s1,8(a5)

1038e: 0107b383 ld t2,16(a5)

10392: 01164633 xor a2,a2,a7

10396: 0106c6b3 xor a3,a3,a6

1039a: 8f29 xor a4,a4,a0

1039c: 01d64633 xor a2,a2,t4

103a0: 01c6c6b3 xor a3,a3,t3

103a4: 00674733 xor a4,a4,t1

103a8: 012648b3 xor a7,a2,s2

103ac: 0096c833 xor a6,a3,s1

103b0: 00774533 xor a0,a4,t2

103b4: 0917b023 sd a7,128(a5)

103b8: 0907b423 sd a6,136(a5)

103bc: ebc8 sd a0,144(a5)

103be: 258d addiw a1,a1,3

103c0: 8616 mv a2,t0

103c2: 86fe mv a3,t6

103c4: 877a mv a4,t5

103c6: 07e1 addi a5,a5,24

103c8: 82f6 mv t0,t4

103ca: 8ff2 mv t6,t3

103cc: 8f1a mv t5,t1

103ce: fb3597e3 bne a1,s3,1037c <sha\_transform+0x42>

**Adpcm:** Adaptive Differential Pulse Code Modulation (ADPCM) is a variation of the well-known standard Pulse Code Modulation (PCM). A common implementation takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1. The input data are small and large speech samples.

For this application, almost all the applications have the same dynamic icount. The only major difference is between RISCV32 vs. ARM32/MIPS32. Following shows the part of the code that causes this difference:

ARM:



RISCV:



The main difference here is the way the branch is handled where for only one RISCV beqz instruction, there are five ARM instructions.

**CRC:** This benchmark performs a 32-bit Cyclic Redundancy Check (CRC) on a file. CRC checks are often used to detect errors in data transmission. The data input is the sound files from the ADPCM benchmark.

There are a few interesting observations for this benchmark. First, similar to Blowfish, RISCV requires notably lower number of instructions due to a difference in the *\_IO\_getc* function in RISCV glibc, where the code first checks whether the file needs to be locked or not and then if required called the lockfile routine, and in this case, it saves a lot of instructions since locking is not necessary (same as Blowfish). Also, here again, combing shift and load into one instruction enables ARM to save three instructions in the main loop of the CRC. Following shows the code for the main loop:

ARM:



**RISCV:**

****

As seen in this code, using *ldr x0, [x21, x2, lsl #3]*, eliminates the need for a separate *lsli* instructions in ARM. Furthermore, using an extra register (x21), eliminates the need for adding an offset (s2) to the base register (i.e. for RISCV: *add a5,a5, s2)*.

**FFT**: This benchmark performs a Fast Fourier Transform and its inverse transform on an array of data. The input data is a polynomial function with pseudorandom amplitude and frequency sinusoidal components.

The big difference in this benchmark is the way GNU MP (multiple precision) libraries are implemented (e.g. mpn\_mul, mpn\_div, etc.). Following shows the code for \_\_mpn\_mul for ARM and RISCV respectively:



RISCV



As seen in these codes, ARM compiler manages to reduce the number of instructions especially by utilizing instructions like *cset* and *csel* and combining load and shift *ldr* and add and shift *add/lsr*.

# Statistics from GEM5

In this section we provide results obtained by well-known cycle-accurate simulator gem5. Using gem5 enables us to find runtime statistics (e.g total number of cycles) and micro-architecture related statistics (e.g. cache miss rate).

For gem5, we use same 11 applications from MiBench benchmark suite. For running these applications, we had to limit the runs to only two available architectures: ARM64 and RISCV641. For each of these two architectures we use a simple in-order processor and a more sophisticated out-of-order core. We use the exact same binaries and inputs used in QEMU.

We first checked the total number of instructions reported by gem5 and cross checked them with dynamic icount numbers in QEMU. Table below shows this comparison for ARM-gem5 and ARM-qemu. As seen in this table, total number of dynamic instructions have only 0.2% difference on average with only 0.5% maximum difference, which confirms that the total dynamic icount reported by gem5 is very similar to that of in qemu.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | basicmath | bitcount | qsort | susan | dijkstra | blowfish | rijndael | sha | adpcm | crc32 | fft | avg |
| ARM-gem5 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| ARM-qemu | 1.002 | 1.004 | 1.005 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1.004 | 1.002 |

Similarly, for RISCV, a same comparison is made where in that case, too, the difference was about 0.3% on average.

Next figure shows the total number of cycles for RISCV and ARM for both in order and out-of-order processors using 11 applications from MiBench benchmark suite.

As seen in this figure, on average, ARM in-order processor is 13% faster, and for out-of-order ARM is about 16% faster than RISCV. Interestingly, in out-of-order cores “blowfish” and “crc32” are significantly faster in RISCV.

Next figure shows the Instruction-Per-Cycle (IPC) for these two architectures. As seen in these two figures, ARM in-order core is about 1.14x faster than RISCV on average. For OoO, ARM is 1.42x faster than RISC. Note that this is irrespective of total number of cycles and mostly depends on micro-architecture features/behaviors in each of these cores. In the next page we provide some more details on why these two architectures are different and provide explanations for outliers in IPC and total number of cycles for RISCV and ARM.

## Outliers in gem5 Statistics:

To find the micro-architectural differences between ARM and RISCV several different statistics including branch rate, branch predictor accuracy, total accesses to memory, average memory bandwidth, number of stall cycles, total number of squashed instructions, cache miss rate, cache latency, and functional unit occupancy are checked. Among those we found that the root cause for different IPC is coming from number of branches and branch predictor accuracy.

One of the major differences between RISCV and ARM statistics is the number of *indirect branches*. Interestingly, while the total number of branches for both architectures on average is almost the same, RISCV has significantly larger number of indirect branches (*JALR).* Looking deeper we find thatRISCV compilertends to use more function calls and more indirect branches for those function calls than that of in ARM (shown below).Furthermore, the current default gem5 branch predictor (a tournament branch predictor with an indirect branch predictor) does a very poor job on correctly predicting these indirect branches which caused a notably high number of misprediction. These predictions cause lots of wasted cycles due to squashing and replaying which is the main cause for higher IPC for most of the applications (except CRC and Blowfish) in ARM. Followings show the total number of indirect branches for RISCV and ARM OoO processors:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| basicmath | bitcount | qsort | susan | dijkstra | blowfish | rijndael | sha | adpcm | crc32 | fft |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 926 | 285 | 285 | 304 | 2649 | 210 | 263 | 260 | 115 | 250 | 406 |
| 666035 | 3375566 | 3284446 | 32400 | 274604 | 406157 | 405810 | 1470 | 13300 | 6774 | 3512138 |

As shown in this table, on average RISCV has 100x more number of indirect branches to that of in ARM which causes to more than 10x misprediction and 10x number of squashed instructions in RISCV which eventually causes a huge performance overhead to the system.

To improve the performance, either the compiler needs to be fixed to reduce the number of indirect branches, or a better branch predictor (especially designed for indirect branches) should be used.

## CRC and Blowfish

Unlike other applications, CRC and Blowfish have higher IPC in RISCV. Looking further, ARM shows a poor performance on these two applications while RISCV maintains a close to average IPC for these two benchmarks. In other words, the main reason for better IPC in RISCV is due to below average performance of ARM in these two applications.

Looking into different statistics, we find that the main reason for decrease in performance is higher number of dependent load instructions and lower DRAM bandwidth utilizations for these two applications in ARM.

As seen in these two figures, there is a correlation between the IPC and memory bandwidth for all these applications. Recall that for both these applications, a routine in glibc was implemented differently in RISCV than ARM which eliminated the need for calling the *flockfile* when the lock was not required. Eliminating this routine, significantly reduces the number of dependent loads in CRC and Blowfish which results into better memory bandwidth utilizations.

# Results for Dhrystone and Coremark

## Dhrystone

Using gem5 and xt-run, following shows the results for running Dhrystone on ARM64 (in-order and OoO), RISCV64 (in-order and OoO), and Xtensa Fusion G3. Same flags are used to compile all the binaries. We used 1 million iterations for Dhrystone for all the runs. For RISCV and ARM, we matched the cache size and number of caches to that of in Xtensa. 8w and 4w stands for 8-wide and 4-wide processors.

Following shows the results for Dhrystone on MIPS, ARM, and RISCV (both 32 and 64bit) using QEMU. The results are normalized w.r.t RISCV64. Dhrystone with 1 million iterations is used for all these runs. We used the number reported by the application itself as the DMIPS. Note that while this number is smaller that the actual hardware, normalized value is still a valuable information since all the architectures are slowed down by same factor.

## Coremark

Using gem5 and xt-run, following shows the results for running Coremark on ARM64 (in-order and OoO), RISCV64 (in-order and OoO), and Xtensa Fusion G3. Same flags are used to compile all the binaries. For all these runs, Coremark with 1 thousand iterations is used. The configs in gem5 are matched to that of in Xtensa.

Following shows the results for Coremark on MIPS, ARM, and RISCV (both 32 and 64bit) using QEMU. The results are normalized w.r.t RISCV64. Coremark with 1 thousand iterations is used for all these runs.

# PPA Analysis

TBC