Comparative Study on Execution Speed & CMR of Various Instruction Set Architectures with gem5

Ashutosh Vishwakarma

April 17, 2025

1 Introduction

In current state, different ISAs are available for systems, some favoured over others. Nonetheless, three major ISAs are `ARM`, `RISCV` and `x86`. Each of them were designed in different time period for different needs. Despite their differences these architectures are slowly being used in comparable machines with most computers still powered with Intel/AMDs `x86` and Macs `ARM`. `RISCV` is gaining space as Linux based OS's like FreeBSD extending to support it. In this study we aim to compare these ISAs on a common benchmark. We'll mostly focus on execution speed with and without caches. We'll be using the `gem5` simulation system for the same. `gem5` provides a huge suite of tools to test the CPU performance. Although we won't be getting into the `C++` core of `gem5`, but we'll be using the `Python` api of `gem5` with the existing modules.

2 Methodology

In this study, we choose the **Sieve of Eratosthenes** as the benchmark algorithm to test on. The Sieve of Eratosthenes is an efficient algorithm to find all prime numbers up to a given number n. It works by iteratively marking the multiples of each prime number, starting from 2, as non-prime. This algorithm is highly efficient for generating a list of primes and has a time complexity of $O(nloglogn)$.

2.1 Compilation

The C code used for the **Sieve of Eratosthenes** is given below:

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
void sieveOfEratosthenes(int n) {
    bool primes[n + 1];
    for (int i = 0; i <= n; i++) {
        primes[i] = true;
    }
    primes[0] = primes[1] = false;
    for (int p = 2; p * p <= n; p++) {
        if (primes[p] == true) {
            for (int i = p * p; i <= n; i += p) {
                primes[i] = false;
            }
        }
    }
    printf("Prime numbers up to %d are:\n", n);
    for (int i = 2; i <= n; i++) {
        if (primes[i]) {
            printf("%d ", i);
        }
    }
    printf("\n");
}
int main() {
    sieveOfEratosthenes(NUM);
    return 0;
}
```

The C code for **Sieve of Eratosthenes** has a macro `NUM`, not defined in the code itself. Rather it will be defined at compile-time by passing compiler args. The same code is compiled to binary for `x86`, `ARM` and `RISCV`. We are simulating the execution in `se` mode of **gem5**, meaning we are not providing complete OS, rather utilizing the host OS for system calls. This creates a challenge for dynamically linked binaries, as the simulation is running on a `x86` machine unable to link for binaries of other architectures. Thus, we compiled the binaries in static linking mode, with `-static` argument. All the binaries are in `64 bit LE`, format as **gem5** currently only loads binaries that are in little-endian format. The compilers used for the compilation are:

Arch	Compiler
x86	gcc
ARM	aarch64-linux-gnu-gcc
RISCV	riscv-linux-gnu-gcc

Table 1: Compilers Used

The compilation command for different binaries are:

```
<compiler> -static -DNUM=<size-of-sieve> -o <output-binary> <input-file.c>
```

2.2 gem5 System Architecture Design

The **gem5** simulation has a modular design. It closely resembles a real system. The main part on which simulation run is a `board`. There are multiple types of boards in **gem5**. A `board` holds different components of a system namely clock, processor and memory. The binary resource is also loaded to this board. The Python api for **gem5** has following definition of a board:

```
board = SimpleBoard(
    clk_freq="1GHz",
    processor=processor,
    memory=memory,
    cache_hierarchy=cache_hierarchy,
)
```

These four components make up the board. The clock frequency in the simulation was set to `1GHz`. The processor used is derived from `SimpleProcessor` class from the **gem5**'s Python api. The `SimpleProcessor` is defined as:

```
processor = SimpleProcessor(
    cpu_type=CPUTypes.TIMING,
    num_cores=1,
    isa=isa
)
```

Here, `CPUTypes.TIMING` is used for better accuracy in time and cycle simulation during execution. The number of cores is set to 1 for all the instances. Finally, the `isa` is varied as per the requirement.
The main memory used in the simulation is a `SingleChannelDDR3_1600` with a size of `1GiB`.

```
memory = SingleChannelDDR3_1600("1GiB")
```

This is kept constant throughout the simulation.
Finally, the `cache_hierarchy` in the simulation is altered across different values. Some system configurations had no Cache at all and other has a two level(L1, L2) cache hierarchy. Also the sizes of the caches are varied.
The summary of different parameters is given below.

Parameters	Values	Description
Architecture	ARM, RISCV, x86	used ISA type
Cache Availability	Yes, No	availability of cache memory
L1DSize	32, 64, 128 kB	size of L1 data cache
L1ISize	32, 64, 128 kB	size of L1 instruction cache
L2Size	128, 256, 512, 1024, 2048 kB	size of L2 cache
Sieve Size	1e1,1e2,1e3,1e4,1e5	size of Sieve of Eratosthenes

Table 2: Parameters of Simulation

2.3 Target Parameters

In this simulation we aim to collect information related to execution performance and delay in various ISAs and configurations. There are 12 total parameters that we aim to collect from the output results. A brief description of the same is given below.

Parameters	Description
`simSeconds`	simulation time in seconds
`simTicks`	simulated ticks
`simInsts`	number of instructions simulated
`simOps`	number of operations simulated
`core.cpi`	cycles per instruction
`core.ipc`	instructions per cycle
`l1d-cache-0.demandHits::total`	total hits to L1D cache
`l1d-cache-0.demandMisses::total`	total misses to L1D cache
`l1i-cache-0.demandHits::total`	total hits to L1I cache
`l1i-cache-0.demandMisses::total`	total misses to L1I cache
`l2-cache-0.demandHits::total`	total hits to L2 cache
`l2-cache-0.demandMisses::total`	total misses to L2 cache

Note: simOps counts all operations including micro-ops

Table 3: Target parameters to be monitored

In further analysis, using the exact counts of hits and misses to caches can produce misleading interpretations. So, we will use *miss_ratio* instead.

$$miss\_ratio = \frac{total\_miss}{total\_miss + total\_hit}$$

One more thing to keep in mind is that,

$$cpi = \frac{1}{ipc}$$

In this simulation, a master python script iterates over the parameters given in Table 2 and runs runs the simulation script by passing the arguments. Each simulation result file is then renamed and moved to separate directory. Considering the number of parameters from Table 2, there are a total of 18 simulations with `No_Cache` and 810 with different `Cache` models, amounting to a total of 828. The result files are then parsed to collect the target parameters mentioned in Table 3 in a CSV file.

3 Results

The collected results give us an insight of the effect of different ISAs and **Cache** models on performance of code execution.

3.1 Execution Performance

The task done by all the ISAs in consideration is same. But the way the C code is translated to executable instruction is different in different ISAs leading different number of instructions needed to be performed to do the same amount of job. The `simInsts` parameters from results report the number of simulated instructions.
From the Figure 1, we can note that `ARM` has to execute minimum number of instructions followed by `RISCV` and `x86`. The simulation was running on a CPU with a clock frequency of `1GHz`, implying keeping other parameters constant `ARM` will perform tasks faster compared to the other two. This observation can be accounted to the simplicity of the instructions in `ARM`. Unlike `x86`, `ARM` has fixed length and less number of instructions and a high code density. Also `ARM` instructions are highly orthogonal, meaning that most instructions can work with any register and support various addressing modes. This reduces the need for special-purpose instructions and makes general-purpose ones more versatile. Although `ARM` and `RISCV` are based on same `RISC` principle various added advantages like conditional execution and dense addressing modes allow for lesser instructions than `RISCV` too.
The trend also follows for the number of opcodes and micro-opcodes executed

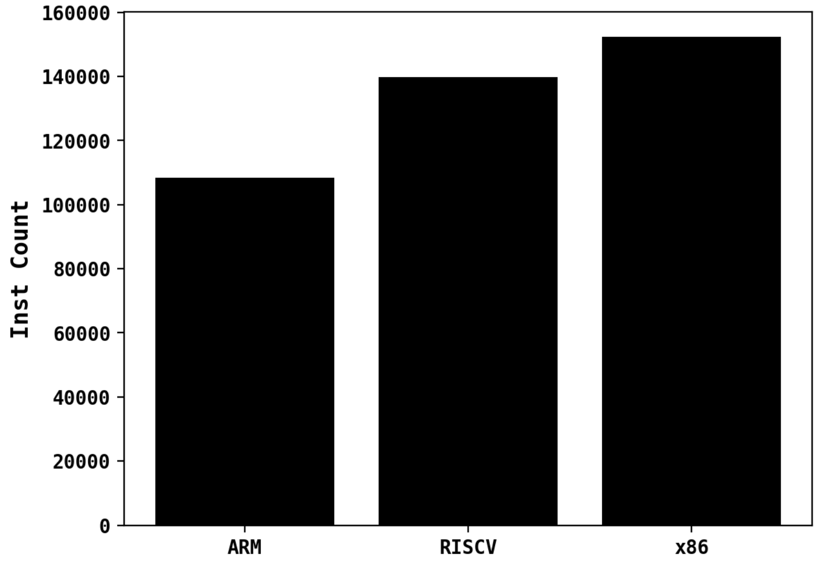

Figure 1: Number of instructions executed by different architectures

(`simOps`) during simulation.
The difference between the number of opcodes executed between `RISC` based architectures and `x86` is stark. `x86` has to execute roughly 2.3 times more opcodes than `ARM`. Although `RISCV` is closer to `ARM` with only 1.2 times more opcodes, highlighting the efficiency of `RISC` based programs.
This further translates into the runtime of the programs. From Figure 3, we can say that `ARM` perform the task faster with a huge margin compared to others. Specifically, `ARM` took 1.8 times lesser time to run than `x86` and 1.6 times less time than `RISCV`. Despite having a lower number of instructions and opcodes the time of execution of `RISCV` is closer to `ARM` than `x86`. This can be counter-intuitive to think of. But next parameter that we looked on explains this result.
As an instruction need not to take only one cycle to execute. In fact most

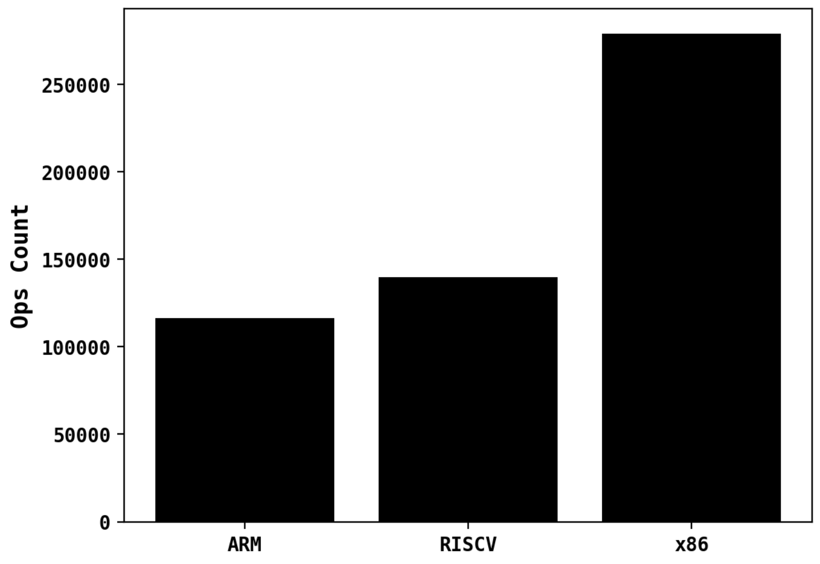

Figure 2: Number of opcodes executed by different architectures

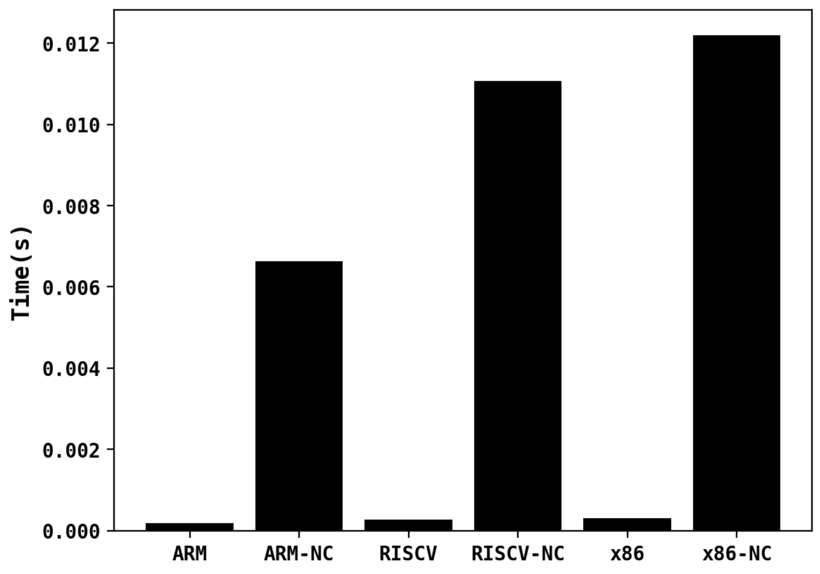

Figure 3: Total execution time

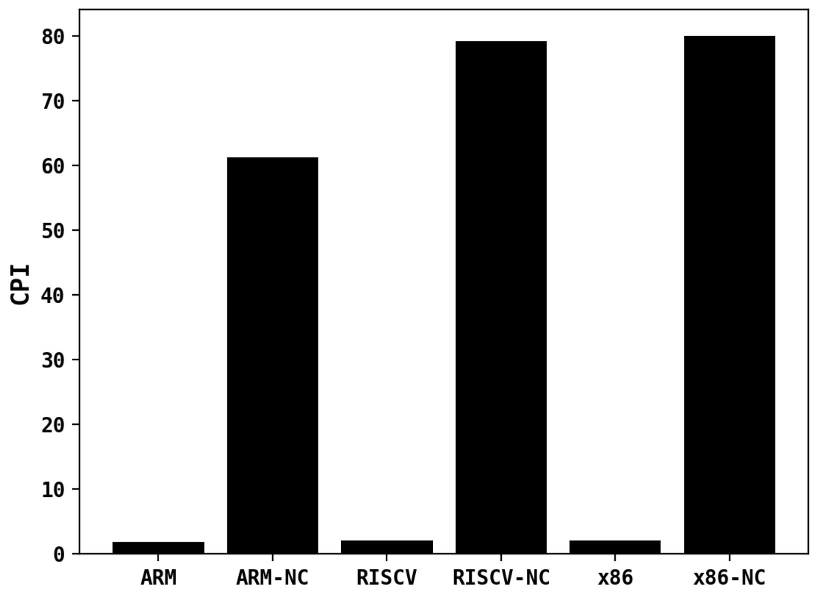

Figure 4: CPI

of the instructions take multiple cycles to run. This is measured in CPI or cycles per instruction. Figure 4, shows the CPI of different architectures. It's evident that RISCV uses more cycles to execute a single instruction compared to ARM. It roughly has similar CPI to x86. Thus, RISCV has a larger execution time than ARM.
Overall, ARM performed operations faster as compared to other two mentioned accounting for lower instructions, opcodes and a lower CPI.

3.2 Cache Performance

Cache works a great deal to improve the performance of the CPU by reducing the bottle-neck of a slower data transfer rate of the main memory. From Figure 3, the difference between the speed of Cache versus No_Cache system is evident. Even the fastest ARM without Cache is much slower than x86 equipped with Cache. Taking the example of ARM, the Cache version is roughly 35 times faster than the No_Cache version of ARM and even Cache version of x86 is roughly 21 times faster than No_Cache version of ARM. All other ISAs represent a similar story.

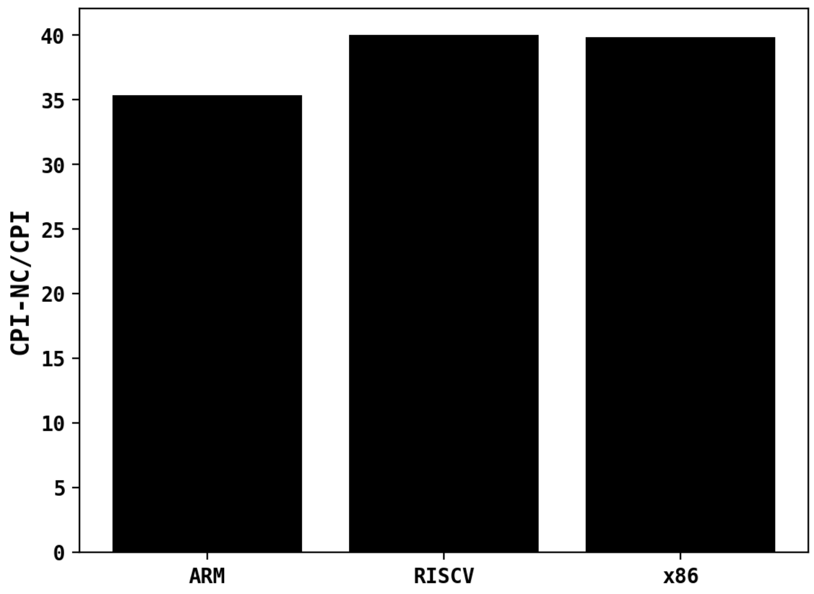

Figure 5: Boost in CPI with addition of `Cache`

From Figure 5, the addition of `Cache` has improved the `CPI` by a factor of at least 30-35. All the previous results discussed, were done by keeping the size of problem i.e, size of sieve constant at 100. This is relatively a small problem size. To check the effect of `Cache` we can increase the problem size.

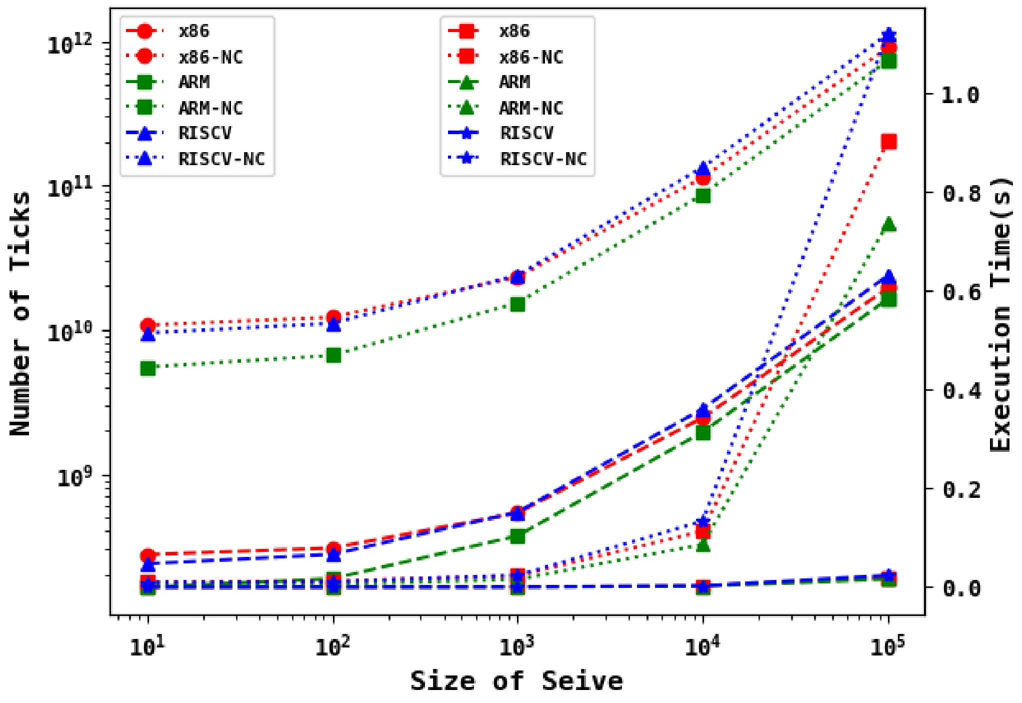

Figure 6: Number of `Ticks` and execution time w.r.t size of sieve

Figure 6 shows some obvious trends that we have discussed earlier, like `No_Cache` version are slower by large factor and `ARM` performs better in both `Cache` and `No_Cache` versions as compared to others. But new interesting observation is that `RISCV` fares slower than `x86` for larger sizes of problem. The gap at the 1e5 between execution time of `RISCV` and `x86` is huge.

The cache miss ratio is a crucial parameter in understanding the performance. Figure 7-9 show the cache miss ratio of various `cache_hierarchy`.

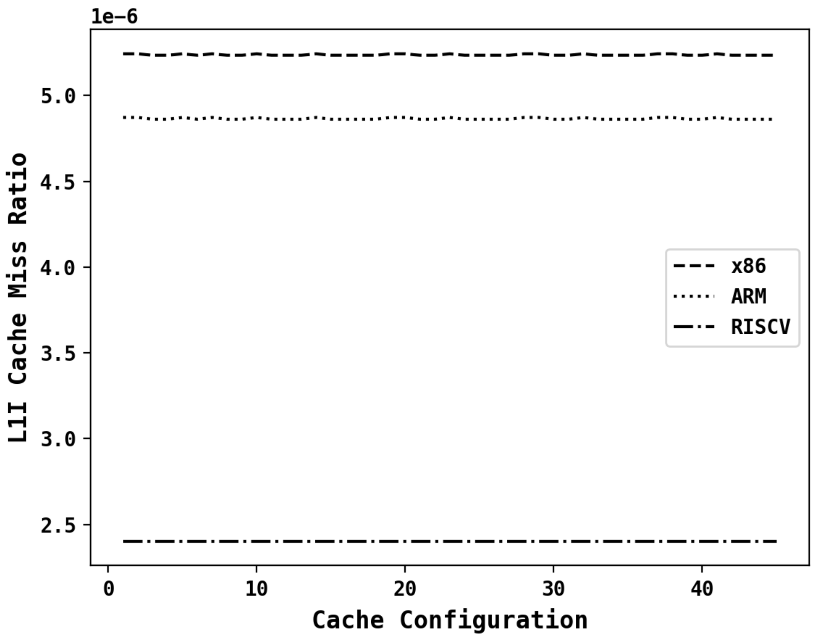

Figure 7: Cache Miss ratio on L1 Instruction Cache

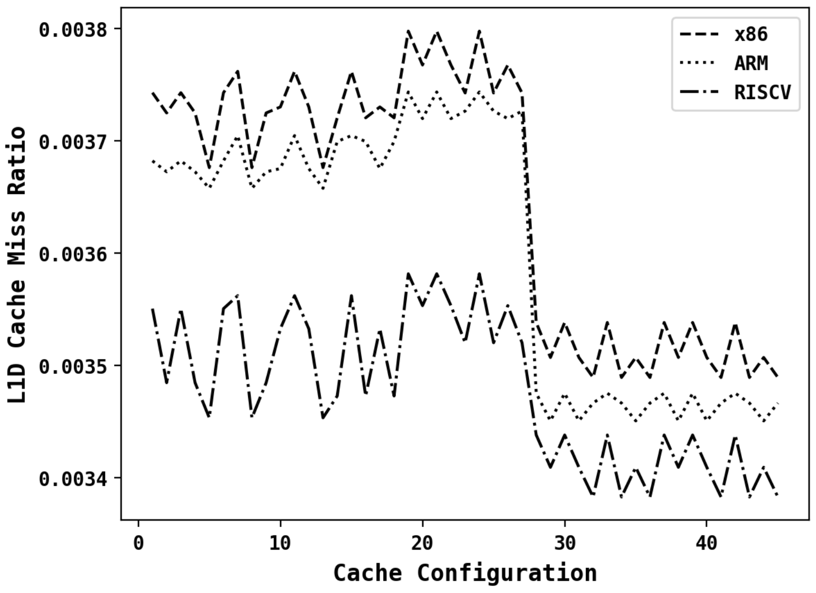

Figure 8: Cache Miss ratio on L1 Data Cache

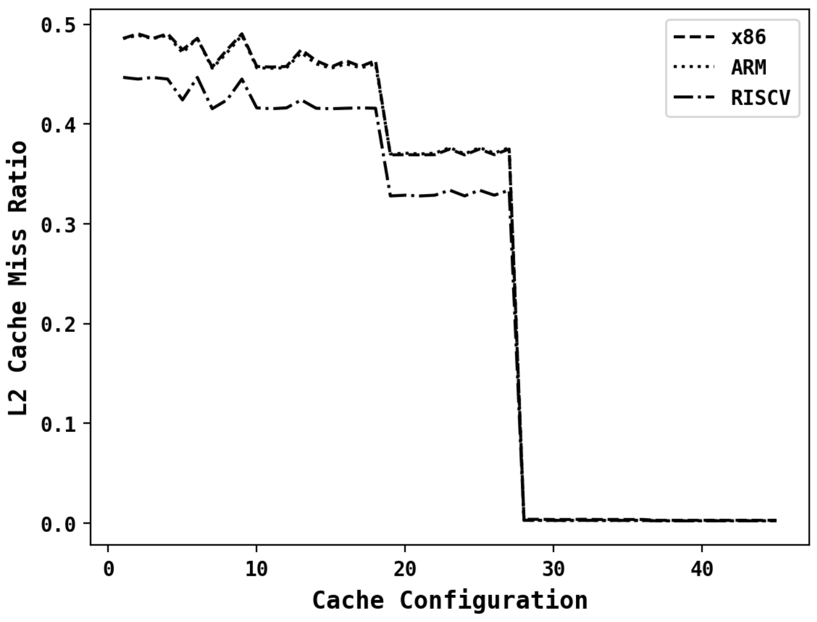

Figure 9: Cache Miss ratio on L2 Cache

From Table 2, there are three parameters affecting the `cache_hierarchy`, namely `L1DSize`, `L1ISize` and `L2Size`. Taking the possible value from Table 2 for each parameters we get a total of 45 configurations. The array all these configurations tuples, is then sorted taking the sum of total cache as key. The index of these configurations in the sorted array is used to plot the corresponding results in figures 7-9. For smaller cache sizes `ARM` and `x86` tend to perform similar while `RISCV` performs best. This trend continues for the `L1D` cache for larger cache sizes as well. But for `L2` cache all the architectures perform similar for larger cache sizes.
Figure 6 shows the cache miss ratio of the `L1I` Cache, across the architectures. Since, total number of instructions were pretty small a larger cache size doesn't affect the miss ratio. On the other hand from Figure 8 and 9 the increase in cache size leads to significant drop in miss ratio. Specifically in case of the `L2` cache the miss ratio drops to 0 for large cache memory. In case of the `L1D` cache the increase in size although drops the miss ratio but that is not much significant. Major impact of the size of the cache is seen on the `L2`'s miss ratio. Although, the effect can be attributed to the huge size increment to `L2` from a mere `128kB` to `2048kB`. Nonetheless, all architectures

perform close on the cache miss ratio. The major factor affecting it is size not the architecture itself.

4 Conclusion

From the study above, it can be concluded that in light of a simple algorithm of **Sieve of Eratosthenes** `ARM` performs best among the three in terms of speed. It can be credited to the simple yet dense instructions of `ARM`. Also `RISCV` being from the same `RISC` family as `ARM` performs closer to `x86`. The cache memory however plays a very significant role in improving the execution time but it does so for all the ISAs alike. The cache miss ratio discussed is affected most by the size of the cache than ISA. However, different ISAs produce different cache miss ratio but the difference is not significant enough to favour one over other.