Discerning Performance, Power, Energy and Area Efficacies of Democratized ISA Effort

ABSTRACT

Instruction Set Architecture (ISA) is fundamental to how a wide variety of modern computer systems – ranging from simple handheld mobile devices to large scale data centers and server farms – are conceived, designed and implemented. ISA designers' goal is often to capture the most basic functions and tasks that can be then used as basic building blocks to compose and express complex applications and softwares. The general expectation is that a computing system should perform functions captured by ISA set in most efficient manner in terms of performance, power and energy.

While an ISA is central to computer design, there have been only a handful of successful ISAs till date. This limits designers' options and forces them to rely on a small subset even though it might not be efficient in capturing necessary functions needed for higher level applications. Unlike compilers, OSs, drivers, and other software components, ISAs have been a proprietary component by-and-large.

Democratization of ISA was the main theme behind the advent of RISC-V. It was touted to relieve the designer community and small- to mid-scale OEMs from the clutches of proprietary ISA suppliers. While this is a novel thought in spirit, in reality, much depends upon the efficacies of democratized ISA itself. In this work, we set out to discern and quantify the viability of an open source ISA such as RISC-V. We conduct numerous program and microarchitectural analysis on major RISC ISAs (ARM, MIPS and RISC-V) and present our findings in the paper. Overall, there is still a lot needs to be done to fully democratize ISA.

Keywords: RISC-V, ARM, MIPS, Performance, Power, Area, Energy, Program analysis, Microarchitecture.

1. INTRODUCTION

In past few decades, numerous ISAs have been proposed and designed but only a handful of them proliferated to make a real impact. In the long series of ISA invention, RISC-V is an emerging ISA and is increasingly becoming an important option for both academia and industry when considering new microprocessor designs [**?**]. Features like modularity, extensibility, simplicity, and being open and free to use, make RISC-V an attractive option for next generation of processors especially in embedded systems domain where new, customized, low-power, and efficient cores are needed.

ISA has a key role in designing cores for different domains. x86 ISA has become dominant in desktop and server domains [**?**], and ARM has become the dominant ISA in mobile, tablet, and embedded system domain [**?**]. The question of impact of ISA design on different Performance, Power, Area (PPA) metrics has traditionally been an important concern for designers and semiconductor industry especially in the 1980s and 1990s when chip area and processor design complexity were the primary constraints [24, 12, 17, 7]. In the past decade, radical changes in computing landscape and rise of mobiles, tables and increasing popularity of ARM ISA this question again becomes an important issue.

Today, with proliferation of embedded and cyber-physical systems (e.g. IoTs) and increasing popularity of domain-specific languages and emerging applications like machine-learning and more importantly, introduction of a new, open-source, modular ISA (RISCV), this question once again becomes an interesting topic for research. We show which of the well known metrics are ISA-dependent and what are the other important factors that impact PPA. Using these experiments we pinpoint the shortcomings, issues, and advantages of using RISC-V ISA over ARM and MIPS ISAs.

In this paper, we present a comparative study on impacts of three well-known ISAs (MIPS, ARM, and RISCV) on performance, power, and area (PPA) on state-of-the-art embedded processors through a systematic measurement campaign using several different toolchains and frameworks. In particular, we study the impact of these ISAs on important metrics such as static and dynamic instruction count (*icount*), total cycles to execute a program, microarchitectural statistics (e.g. MPKI, branch prediction accuracy, etc.), dynamic power, and core's area. We report our key findings on impacts of using different ISAs on each of these metrics. We find that some of these are ISA-dependent and other metrics are dependent on other factors such as compiler, runtime libraries, and specific microarchitectural features.

Our primary observation is that while comparing to MIPS and ARM, RISC-V has some shortcomings and design/toolchain issues that should be addressed and fixed to make it more competitive. On the positive side, due to its intrinsic features such as modularity RISC-V provides a great opportunity for designing customized

PPA-efficient cores.

Rest of the paper is organized as follows: Section 2 presents an overview of the background to our research. Section 3 details out methodology. Section 4 presents our experimental setup and detailed results. Finally, section 5 presents an overview of related work before we conclude in Section 6.

2. BACKGROUND

RISC-V is an emerging open-source software and hardware ecosystem that has gained in popularity in both industry and academia [2, 11]. At the heart of the ecosystem, the RISC-V ISA is designed to be open, simple, extensible, and free to use. The RISC-V software tool chain includes open-source compilers (e.g., GNU/GCC and LLVM), a full Linux port, a GNU/GDB debugger, verification tools, and simulators. On the hardware side, several RISC-V prototypes (e.g., Celerity [4]) have been published. The rapid growth of the RISC-V ecosystem enables computer architects to quickly leverage RISC-V in their research.

RISC-V ISA has a modular design and defines parts of its ISA as extensions that are coded with a letter and written as “RV32I” or “RV64IMAFD”. There are small differences in the same extension for different register sizes. If the architecture (32/64 bit) is not important it can be left out and described as ”RVI” or ”RVIMAFD”. The ISA Base and its extensions are developed in a collective effort between industry, the research community and educational institutions. Extensions marked as ”frozen” are not expected to change in any way except clarifications and improvements in documentation.

The base set of RISC-V is the Base Integer Instruction Set ”RV32/64/128I” or ”RV32E” (a reduced version of RV32I that supports only 16 registers designed for embedded systems). This set by itself can implement a simplified general-purpose computer, with full software support, including a general-purpose compiler.[4]

A computer design may add further extensions: Integer multiplication and division (”M”), Atomic instructions (”A”) for handling real-time concurrency, IEEE Floating point (”F”) with Double-precision (”D”) and Quad-precision (”Q”) options.[4] There’s also an optional ”compact” (”C”) extension to reduce code size. Many RISC-V computers might add the compact extension to reduce power consumption, code size, and memory usage.[4]

There are future plans for to support hypervisors, virtualization,[18] bit-manipulation (”B”), decimal floating-point (”L”), Packed SIMD (i.e. budget multimedia, ”P”), vector processing (”V”) and transactional memory (”T”).[4] Below is an overview with all extensions that are finalized (frozen) and currently in development.

A small 32-bit computer for an embedded system might be ”RV32EC”. A large 64-bit computer might be ”RV64IMAFDC”. A computer with the instruction sets ”IMAFD”, an ”RV32/64IMAFD”, is said to be ”general-purpose”, summarized as ”G,”[4] so an ”RV64IMAFDC” can also be described as an ”RV64GC”. Together with the ”privileged” instruction set extension an ”RVGC” defines all instructions needed to conveniently support a Unix-style operating system.

3. METHODOLOGY

To study the effects of ISAs on Power, Performance, and Area, we used several different metrics using 4 different tools and more than 12 standard benchmark applications. Followings describe the frameworks, metrics, and benchmarks used in this paper. The reader can skip this section if he/she is uninterested in these details.

3.1 ISAs and Compiler

Table 1 shows the ISAs used in study. For each of these ISAs, we use *gnu-gcc* cross-compiler. We intentionally chose gcc so that we can use the same front-end to generate all binaries. All target independent optimizations are enabled (O3); machine specific tuning is disabled so there is a single set of binaries for each ISA. For MIPS, *-march* is used to generate MIPS release 6, 32bit and 64bit versions. For ARM, two separate compilers ARMv7 and AARCHv8 are used to generate 32bit and 64bit ARM binaries. Finally, for RISCV we use the gnu-toolchain provided by RISCV developers publicly available in github. We believe using similar flags and front-end could help us to mitigate the effect of compilers on performance and power metrics, however, we will later show that RISCV compiler does have important inefficiencies and issues that could hurt the performance of the system.

Table 1: ISAs used in this study.

ISAs	Specification
ARM	32v7, 64v8 (AARCH)
MIPS	32r6, 64r6
RISCV	rv32g (IMAFD), rv64g

3.2 Framework

3.2.1 QEMU

To find the dynamic and static instruction count (ICOUNT), we use a well-known open-source emulator called Quick-EMUlator (QEMU). QEMU is a hosted virtual machine monitor: it 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. We chose QEMU primarily cause it can emulates MIPS, ARM, and RISCV ISAs in user-mode.

Static Instruction Count: Static icount is a classic metric to show the code density of different ISAs which can directly affect the performance, power, and area (i.e. required icache size). While static icount can simply be measured by measuring the lines of assembly code in a binary, a more meaningful and useful way to measure this metric is to count number of unique PCs (each PC represents an instruction) seen during the execution of the an application (i.e. counting only those instructions that are actually executed at least once).

We believe this approach provides a better insight on the actual instruction memory footprint and shows the difference between ISAs better. We found that these two number (our approach vs. measuring the size of the code) could be quite different for some applications since compilers might include source codes for all routines in an included library, while some of these routines may not be used at all.

To find static icount, we modified QEMU's source code to add a new data structure to track and count unique PCs during the execution. These changes are made in a routine called EXEC_CPU() which is used in all ISAs. To check the correctness of our model, for each ISA, we used several synthetic benchmarks and checked the QEMU's output to the actual static icount computed manually.

Dynamic Instruction Count: Similar to static icount, dynamic icount is also an important metric to show the runtime behavior of ISAs. Dynamic icount can directly affect total runtime especially in simple in-order cores where Instruction Per Cycle (IPC) for these cores are mostly close to 1 thus the total runtime is determined by dynamic icount metric.

Dynamic icount is also computed in QEMU by modifying the source code to be able to count this metric during execution. The results are validated using a synthetic benchmark where number of iterations of a simple loop changed in different runs. We checked whether QEMU correctly reports dynamic icount as the number of iterations changed for each run.

Per-PC Iteration Count: Another interesting metric which used in this paper is per-PC iteration count where for each PC we report how many times this instruction has been executed. This could be very useful to find the hot regions in the code, and find the reason(s) behind why some applications have significantly higher/lower dynamic icount. More details will be shown in the Section ??. QEMU is modified to count this metric too, during execution.

3.2.2 gem5

gem5 is a well-known, open-source, cycle-accurate simulator which can simulate in-order and out-of-order cores, memory systems, and interconnect in details. Using gem5 enables us to find runtime statistics (e.g total number of cycles) and micro-architecture related statistics (e.g. cache miss rate). gem5 supports ARM and very recently RISCV. Unfortunately gem5 only supports an old version of MIPS and hence we removed the analysis for MIPS in gem5. To have a fair comparison, processor and memory system configurations (i.e. clock rate, issue width, cache levels and size, delays, etc.) are matched for both ARM and RISCV processors. We use a simple single-issue, 4-stage pipeline with no prefetcher and a single-level cache as an in-order core for RISCV and ARM, and use a more sophisticated 4-issue, out-of-order, with a direct/indirect prefetcher and two level caches as an out-of-order core in our experiments. Detailed for these two cores are shown in Table 2.

Table 2: Simulation configuration for gem5.

Parameters	In-order core	Out-of-Order core
Issue Width	1	4
Private Caches	I$/D$ 32KB	I$/D$ 32KB
Shared Caches	N/A	L2 256KB
BranchPred	Tournament BP	Tournament BP
Prefetcher	N/A	stride/next-line

Cycle Count and IPC: we use the number reported by gem5 for these metrics. Same input is used for both RISCV and ARM and the numbers reported are from beginning to end of each application. We use IPC to show the effect of using different ISAs on processor's computation speed. Our key findings about these two numbers will be shown in Section ??.

Microarchitecture Statistics: to gain some insights about the runtime behavior of each core and ISA's impacts on them, we check several microarchitecture-related metrics such as: cache miss rate (MPKI), branch predictor accuracy, instruction mix, fetch and commit rates, total stall/idle/squashed cycles, memory bandwidth utilization, direct/indirect branches and function calls, and dependent memory load/stores across different applications.

3.2.3 McPAT

McPAT is an integrated power and area modeling simulator. It uses ITRS roadmap models at circuit level to model both static and dynamic power of the system. McPAT uses an XML-based interface to read microarchitecture-related statistics generated by a cycle-accurate simulator (e.g. gem5) as inputs and uses a detailed model for cores, memory system, NoC, etc. to estimate the area and power of the system. We use a set of python and shell scripts to parse the statistics from gem5 and fill in the XML template in McPAT.

Dynamic Power and Total Energy: using McPAT, we measure the dynamic power consumption of core (both in-order and out-of-order) and memory system. Using the data from gem5, we also calculate the overall energy for these ISAs.

3.3 Applications

We use a representative set of applications from a standard open-source embedded system's benchmark suite called MiBench. The MiBench suite is commonly used to evaluate the performance of processors intended for the embedded/IoT market, and it was designed to be representative of the computation that is needed in that market (e.g. automotive, industrial systems, etc.). These applications are mostly compute intensive and designed such that have minimum interactions with outside the processors. Many of the applications are also overlapped with EEMBC benchmark suite.

Our applications we picked ranges from basic math abilities (*basicmath*), bit manipulation (*bitcount*), simple data organization (*qsort*), a shape recognition program (*susan*), shortest path calculations (*dijkstra*), to data encryption, decryption and hashing (*blowfish, ri-*

jandael, sha), and communications applications (*fft, crc32, adpcm*). In addition to these 11 applications picked from MiBench, we use two well-known open-source application which often used in industry to report the performance of the processor *Coremark* and *Dhrystone*. For each of these applications the "large" dataset is used. For Coremark and Dhrystone, iterations are chosen such that the total number of executed instructions be more than 200 million instructions (we chose 1000 iterations for Coremark and 1 million iterations for Dhrystone). For each application, same inputs are used for all runs across different ISAs/processors.

3.4 Limitations and Challenges

Clearly our study is very dependent on correctness, accuracy, and availability of several different tools. During this study, we encountered infrastructure and system challenges including library dependencies for cross-compilation, library support in QEMU and gem5 as well as ISAs availability in these tools (e.g. MIPS for gem5). Furthermore, this study becomes even more challenging due to very dynamic state of RISCV's toolchain and ecosystem development where still several features both in ISA itself and its toolchain is evolving. Throughout our work, we focus on what we believe to be the first-order effects for performance, power, and energy and feel our analysis and methodology is rigorous. We believe we did our best to compare these architectures fairly by using same set of tools and using same applications with same inputs and same default flags for compilations. Other more detailed methods may exist, and we will make the data, toolchains, and scripts publicly available to allow interested readers to pursue their own detailed analysis.

4. EXPERIMENTAL ANALYSIS

In this section we report our results and findings of static and dynamic instruction count, performance, power, and energy.

4.1 Instruction Count and Mix

Figure 1 shows the static icount obtained from QEMU. As seen in the figure, on average 32bit and 64bit ARM are about 15% more dense than that of in MIPS and RISCV. Mibench applications have on average 5k instruction that are executed at least once. Figure 2 shows the dynamic icount obtained from QEMU. Unlike static icount, dynamic icount can be quite different from one application to another for different ISAs. However, on average MIPS, ARM, and RISCV have almost same dynamic icount.

Key Findings: 1- Mixed/Combined instructions (e.g. add+shift, mult+add, etc.) and three operand/three-way comparison in ARM could result in significant dynamic and static icount reduction. A possible and interesting extension to RISCV could be adding this sort of instructions to the base ISA (RV-G) for high-performance scenarios. An example is shown in Figure 4, where a same function is shown for ARM and RISCV ISAs. As seen in this example, "ldr" instruction in ARM with embedded "lsl" instruction inside it, has saved one instruction. Further "cmn" (compare and add), also saved two extra instructions in ARM. We found that there are many examples such as this where more complex instructions in ARM could save more space, however, we will later show that this complexity comes with more power consumption.

2- RISCV compiler tends to use many load/store to stack/register files which could be saved by using extra registers.

ARM:

```
400464:  f8627aa0  ldr    x0, [x21, x2, lsl #3]
400468:  91000673  add    x19, x19, #0x1
40046c:  ca5c201c  eor    x28, x0, x28, lsr #8
400470:  aa1403e0  mov    x0, x20
400474:  94002261  bl     408df8 <_IO_getc>
400478:  ca000382  eor    x2, x28, x0
40047c:  3100041f  cmn    w0, #0x1
400480:  92401c42  and    x2, x2, #0xff
                          400464 <main+0x74>  //
400484:  54ffff01  b.ne   b.any
```

RISCV:

```
10226:       6398  ld     a4,0(a5)
10228:        405  addi   s0,s0,1
1022a:   00d74d33  xor    s10,a4,a3
1022e:       8526  mv     a0,s1
10230:   039060ef  jal    ra,16a68 <_IO_getc>
10234:   00ad47b3  xor    a5,s10,a0
10238:   0ff7f793  andi   a5,a5,255
1023c:   078e      slli   a5,a5,0x3
1023e:   97ca      add    a5,a5,s2
10240:   008d5693  srli   a3,s10,0x8
10244:   ff3511e3  bne    a0,s3,10226<main+0x66>
```

Figure 4: A code snippet in assembly showing a same function from Mibench benchmark suite for ARM-64 and RISCV-64 ISAs. Differences shown in rectangles.

A better compiler is needed to improve register utilizations to remove unnecessary load/store register instruction and improve code density in RISCV. Moreover, ARM has a more intelligent compiler where it can unroll and/or saves some instructions in a loop body. This could in turn be very beneficial when the loop is hot.

3- Checking some of the outliers we found that runtime libraries could play an important role in dynamic behavior of the system for these applications, and interestingly there were cases that some of glibc/gnu libraries were implemented differently in RISCV. Overall, introduction of RISCV provides a unique opportunity to rethink and re-evaluate many existing runtime libraries

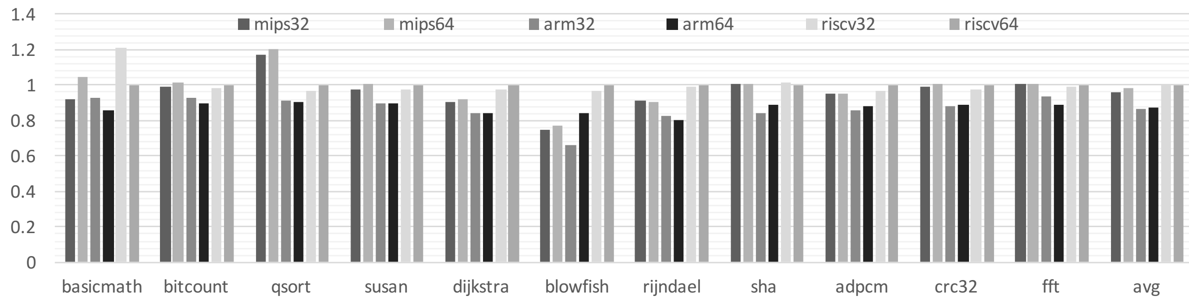

Figure 1: Static Instruction Count for MiBench applications using QEMU. Results are normalized w.r.t. RISCV64. The average number of static instructions is about 5000 for these benchmarks.

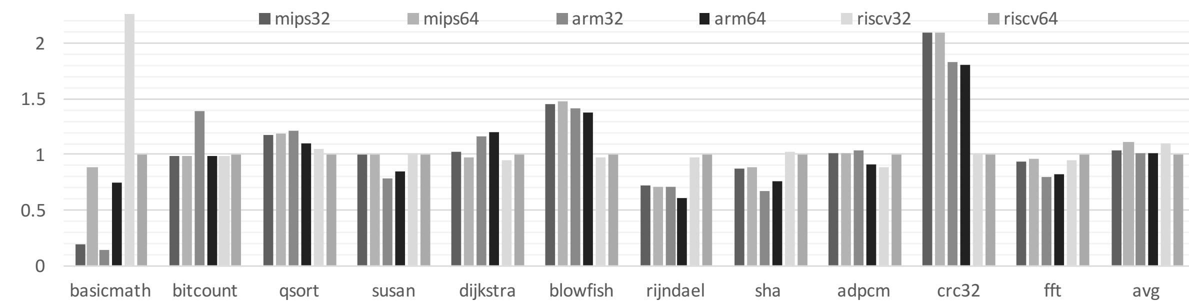

Figure 2: Dynamic Instruction Count for MiBench applications using QEMU. Results are normalized w.r.t. RISCV64. The average number of dynamic instructions for these benchmarks is about 450 million.

which could lead to non-negligible performance/power efficiency improvements.
4- RISCV has a more straightforward way to handle branches (compared to ARM) which could save many cycles in hot loops in some cases.
Outliers: Apart from better density (in static icount) in ARM due to combined/complex instructions, for "basicmath", the main reason for different dynamic icount among ISAs is due to the way *long double* data type is handled (by default) in different compilers. Some used native floating point instructions for mult/div this data type, and the some used software libraries (e.g. *gnu mulf3*). For "crc32" and "blowfish" the main reason for significantly lower icount in RISCV is the way an IO library (part of glibc) is implemented where in RISCV the code first checked whether an atomic lock is needed (shown in Figure 7).

Figure 3 shows the instruction mix for MiBench applications for RISCV and ARM ISAs. The mix is divided into integer/floating point instructions, loads, stores, and branches. As seen in the figure, the instruction mix is ISA-independent to first order and only changes from an application to another.

4.2 Performance

Figure 5 shows the total number of cycles (from beginning to the end of program using same inputs) for 2 in-order and 2 out-of-order cores simulated in gem5. As seen in the figure, overall ARM in-order and out-of-order cores are faster by 5% and 13% over RISCV cores. To further investigate the performance, Figure 6 shows the IPC for RISCV and ARM cores. As seen in this figure, ARM in-order and out-of-order cores have better IPC than RISCV cores by 13% and 42% respectively. Followings show our key findings about IPC and performance:
Key Findings: 1- Looking at different statistics in gem5, we found that the main reason for lower performance in RISCV is that it uses significantly larger indirect jumps and function calls (more than 100x!). This, in turn, leads to a poor branch prediction accuracy and results in a large number of wasted/squashed cycles due to branch mispredictions. We didn't find anything fundamentally wrong with RISCV ISA that could cause this and we believe this problem is mainly due to the RISCV compiler performance. A possible solution to this problem is either fixing the RISCV compiler to reduce the number of indirect branches and improve the inlining or improve the branch predictor accuracy on indirect branches.
2- Apart from indirect jumps, branch predictor accuracy, and number of function calls, almost all other micro-architectural statistics (e.g. MPKI, stalls, etc.) are the same for RISCV and ARM processors.
3- Instruction mix is ISA-independent. Both ISAs have almost the same ALU, LOAD, STORE, and BR instruction distribution.
4- Cycle Count for In-order processors closely follows the dynamic icount thus it can directly be benefited from better compiler and/or custom instructions.

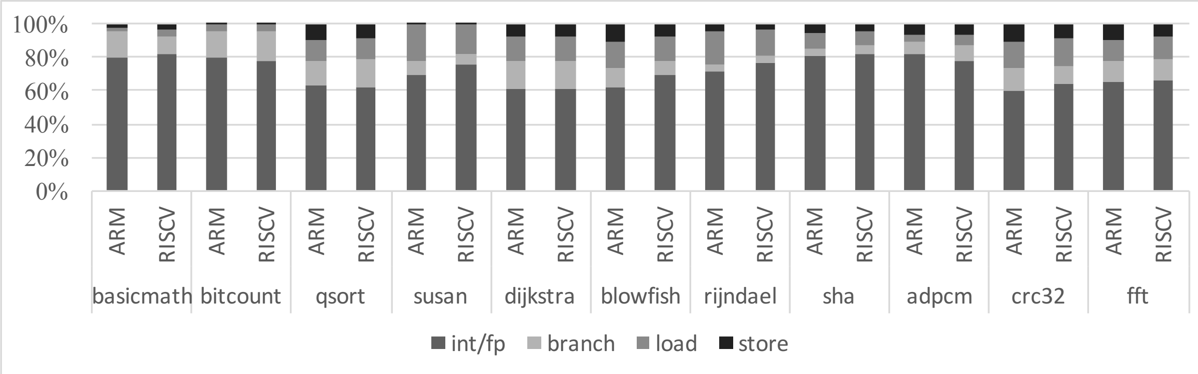

Figure 3: Instruction mix for RISCV and ARM ISAs.

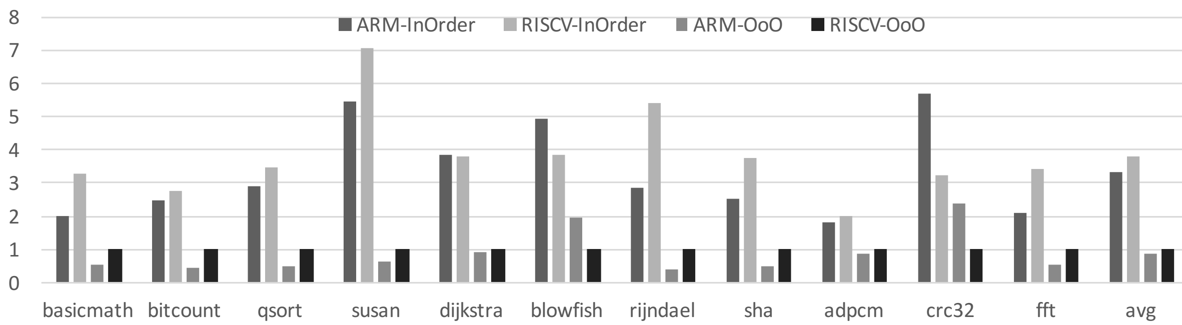

Figure 5: Total number of cycles for RISCV and ARM cores (lower is better). Results are normalized w.r.t. RISCV-Out-of-Order core. The average number of cycles is about 300 million for OoO cores and 1 billion for In-Order cores.

Outliers: "crc32" and "blowfish" have poor performance on ARM processors. Looking into the statistics, we found that this poor performance is mainly due to a large serialized atomic loads to main memory (mainly because of activities in _IO_FLOCKFILE () shown in Figure 7) which causes lots of stall cycles and pipeline underutilization in ARM cores (especially for OoO core).

4.3 Power, Energy, and Area

Figure 8 shows the dynamic power for the ARM and RISCV cores. Figure 9 shows the energy (i.e. power delay product) for these two cores. Overall, while RISCV is more power-efficient in general, the average energy efficiency is almost similar for both cores. In addition for both ARM and RISCV cores, we use McPAT to estimate the area. Results show that in-order cores have almost similar size (about 0.01 mm^2) and out-of-order cores also have very similar size (about 0.2 mm^2). Following shows our findings in more detail:

Key Findings: 1- ARM on average consumes more dynamic power than RISCV (almost 1.34x). The main reason for this larger power consumption is due to more complex fetch/decoding and execution units in ARM.
2- Pipeline under-utilization and stall in two applications ("crc32", "blowfish") caused a significant power consumption reduction in ARM.
3- Power consumption in both in-order and OoO cores for RISCV and ARM follow a very similar trend.

Outliers: Similar to results for IPC, the main reason for low power consumption in "crc32" and "blowfish" on ARM processors is the pipeline underutilization due to many dependent loads in those applications. Moreover, other than "crc32" and "blowfish", RISCV is more energy efficient"dijkstra". The main reason for this is notably lower number of dynamic instructions for RISCV which is due to extra overhead in handling branches in hot loops (i.e. two instructions in ARM "cmp, br" vs. one instruction in RISCV "br").

4.4 Dhrystone and Coremark

We use our framework to also investigate static and dynamic behavior of ARM and RISCV cores using these two benchmarks. Figure 10 shows the total dynamic icount (for same number of iteration) for Dhrystone and Coremark benchmarks (using default, out-of-the-box compiler flags). Dhrystone and Coremark iterations are chosen such that each run executes at least 200 million instructions.

Figure 11 shows total cycles (from beginning to end) for Coremark and Dhrystone applications on RISCV and ARM in-order cores with and without a branch-predictor.

Key Findings: 1- Coremark and Dhrystone follow the same trend as that of in the average dynamic icount

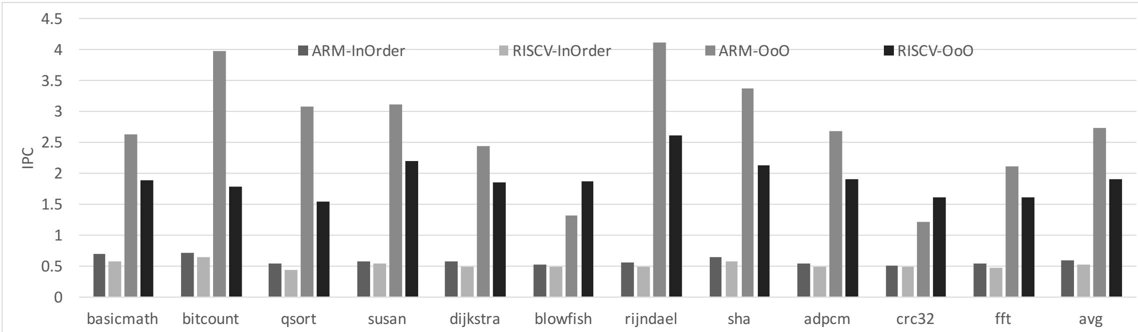

Figure 6: Instruction per Cycle (IPC) for RISCV and ARM out-of-order cores.

ARM and MIPS:

```
int
_IO_feof (fp)
_IO_FILE* fp;
{
int result;
CHECK_FILE (fp, EOF);
_IO_flockfile (fp);
result = _IO_feof_unlocked (fp);
_IO_funlockfile (fp);
return result;
}
```

RISCV:

```
int
_IO_feof (_IO_FILE *fp)
{
  int result;
  CHECK_FILE (fp, EOF);
  if (!_IO_need_lock (fp))
    return _IO_feof_unlocked (fp);
  _IO_flockfile (fp);
  result = _IO_feof_unlocked (fp);
  _IO_funlockfile (fp);
  return result;
}
```

Figure 7: A code snippet in C showing a same IO subroutine implemented in glibc for ARM, MIPS, and RISCV. Differences shown in red.

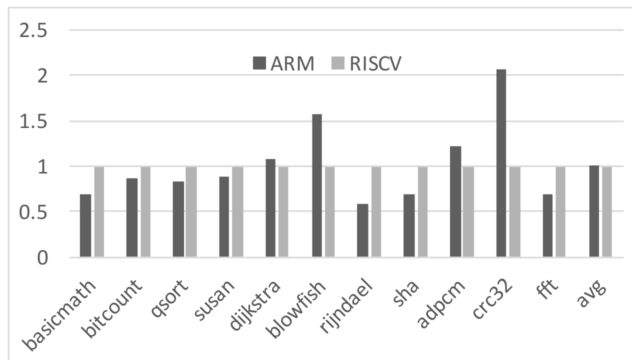

Figure 9: Energy for RISCV and ARM out-of-order cores (normalized w.r.t. RISCV).

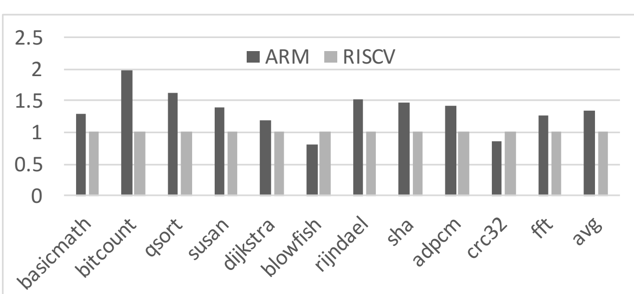

Figure 8: Dynamic Power for RISCV and ARM out-of-order cores (normalized w.r.t. RISCV).

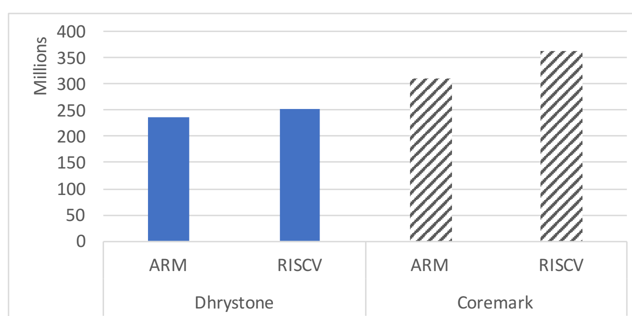

Figure 10: Coremark and Dhrystone dynamic instruction counts for RISCV and ARM ISA.

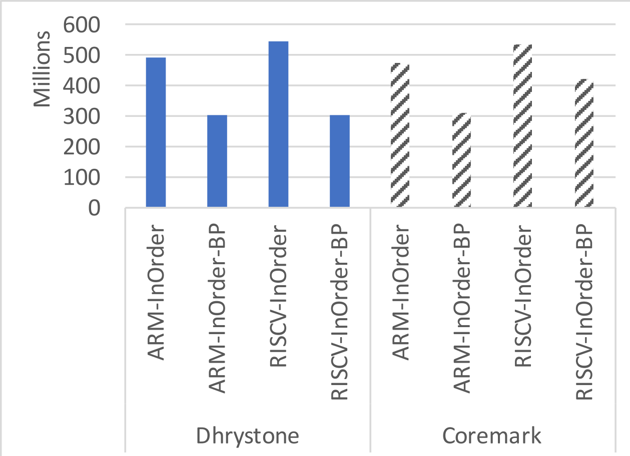

Figure 11: Dhrystone and Coremark cycle count (lower is better) for RISCV and ARM in-order cores (with and without branch predictor).

and total cycles in MiBench applications (i.e. ARM has lower dynamic icount and better performance with almost similar ratio to that of in MiBench).

2- Using a branch predictor has a notable impact on performance of the ARM and RISCV in-order cores.

5. RELATED WORK

Early ISA studies are instructive but miss key changes in todayâĂŹs microprocessors and design constraints that have shifted the ISAâĂŹs effect. We review previous comparisons in chronological order and observe that all prior comprehensive ISA studies considering commercially implemented processors focused exclusively on performance. Bhandarkar and Clark compared theMIPS and VAX ISA by comparing theM/2000 to the Digital VAX 8700 implementations [Bhandarkar and Clark 1991] and concluded: âĂIJRISC as exemplified by MIPS provides a significant processor performance advantage.âĂİ In another study in 1995, Bhandarkar compared the Pentium-Pro to the Alpha 21164 [Bhandarkar 1997], again focused exclusively on performance and concluded: âĂIJthe Pentium Pro processor achieves 80% to 90% of the performance of the Alpha 21164... It uses an aggressive out-of-order design to overcome the instruction set level limitations of a CISC architecture. On floating-point intensive benchmarks, the Alpha 21164 does achieve over twice the performance of the Pentium Pro processor.âĂİ Consensus had grown that RISC and CISC ISAs had fundamental differences that led to performance gaps that required aggressive microarchitecture optimization for CISC that only partially bridged the gap. Isen et al. [2009] compared the performance of Power5+ to Intel Woodcrest considering SPEC benchmarks and concluded that x86 matches the POWER ISA. The consensus was that âĂIJwith aggressive microarchitectural techniques for ILP, CISC and RISC ISAs can be implemented to yield very similar performance.âĂİ Many informal studies in recent years claim the x86âĂŹs âĂIJcruftyâĂİ CISC ISA incursmany power overheads and attribute the ARM processorâĂŹs power efficiency to the ISA.1 These studies suggest that the microarchitecture optimizations from the past decades have led to RISC and CISC cores with similar performance but that the power overheads of CISC are intractable. In light of the ISA studies from decades past, the significantly modified computing landscape, and the seemingly vastly different power consumption of RISC implementations (ARM: 1âĂŞ2W, MIPS: 1âĂŞ4W) to CISC implementations (x86: 5âĂŞ36W), we feel there is need to revisit this debate with a rigorous methodology. Specifically, considering the multipronged importance of the metrics of power, energy, and performance, we need to compare RISC to CISC on those three metrics. Macro-op cracking and decades of research in high-performance microarchitecture techniques and compiler optimizations seemingly help overcome x86âĂŹs performance and code-effectiveness bottlenecks, but these approaches are not free.

6. CONCLUSIONS

To best of our knowledge, this is first work to compare RISC-V with its popular proprietary counterparts (ARM and MIPS). We used state-of-art simulation and emulation frameworks: *qemu* for program analysis and *gem5* for microarchitectural simulations. We also present concrete cases where RISC-V clearly falls behind compared to ARM, MIPS ISAs and what addendum could possibly make it competitive. To our surprise, we also stumbled upon cases where RISC-V is better than proprietary ISAs. Overarching goal of our exploration is to enable RISC-V designers so that they can augment their designs and be able to close the gap with other state-of-art ISAs.

7. REFERENCES