ISAMod: A Tool for Designing ASIPs by Comparing Different ISAs

Shubhankar Suman Singh
Computer Science and Engineering, IIT Delhi, India
Email: shubhankar@cse.iitd.ac.in

Smruti R. Sarangi
Usha Hasteer Chair Professor, Computer Science, IIT Delhi, India
Email: srsarangi@cse.iitd.ac.in

***Abstract*—Designing the ISA (instruction set architecture) is a very critical activity in the entire ASIP (application-specific instruction set processor) design process. There is a long history of using automated tools that suggest custom instructions based on an analysis of the data flow graphs (DFGs) of target programs. Such approaches often create an ISA that is overspecialized for a small set of applications and they often suggest a plethora of custom instructions that cannot be practically implemented. A survey of recent work indicates that adding custom instructions to freely available ISAs such as RISC-V still relies on bespoke analyses and institutional memory. In this paper, we focus on such modern applications, where we only need to add a few instructions to an existing ISA such as RISC-V. The aim is to either supplant or complement the extensive manual analysis that goes into such decision making.**

We propose an unconventional approach that uses novel visualization techniques to first understand the impact of different ISA features by comparing the execution of the same program using different popular ISAs: both RISC and CISC. Our novel graphical methods provide simple and intuitive explanations for differences in performance across ISAs for the same micro-architecture. Moreover, we can use this information to pick desirable instructions from other ISAs and evaluate their impact when they are incorporated. We show examples where we are able to increase the average performance by 16.5% for 6 SPEC-2017 benchmarks by just adding 2-10 extra instructions in the basic RISC-V ISA. The performance gain is comparable with state of the art custom instruction generators. Our tool, *ISAMod* (ISA Modify), achieves this using a very simple and intuitive approach. It has the potential to prove itself as a vital part of the overall design flow and can reduce reliance on institutional memory to a large extent.

***Index Terms*—ASIP design, ISA-comparison, x86, ARM, RISC-V**

I. Introduction

Generation[1] of custom instruction sets for processors that are tailored for a particular class of applications, is a classical problem in EDA and embedded systems [1], [2]. This is a rather old problem, which saw its heyday roughly a decade ago. In those days, the primary focus was to create a bespoke processor with its own custom instruction set that is specifically tailored for executing a class of applications such as media encoders. During this period, hundreds of ideas were proposed for automatically identifying instructions from the dataflow graph (DFG) of the programs' execution [3], [4]. These automated approaches roughly had a similar structure. The first phase was subgraph enumeration, where all candidate subgraphs of the DFG were enumerated – the idea is to replace a candidate subgraph with a single instruction. The next step was subgraph selection, where a subset of the enumerated subgraphs where selected and replaced with custom instructions. In many proposals, this step was followed by assessing the costs of the custom instructions in terms of the original circuit area, the effects of latency, the potential performance benefits, and the additional power consumption. This process finally led to a set of custom ISA extensions. This area

[1]This paper's main focus is a new visualization system to detect performance issues. Hence, to understand this paper, it is necessary to either read a color printout or read it using an electronic device

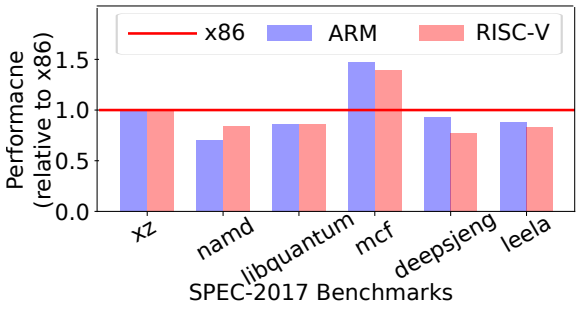


Fig. 1: Performance of ARM and RISC-V ISAs (relative to x86)

is still active – the target applications and the techniques have changed. Researchers nowadays are targeting AI and cryptography-based applications [1], [5]. Furthermore, graph-based techniques have given way to AI-based techniques such as genetic algorithms and Ant colony optimisation [6], [7].

The reason that we revisit this problem in 2020 is primarily because the interest in ISA extensions has been rekindled based on some recent developments. The biggest driver of this new trend is the advent of freely usable instruction sets such as the RISC-V ISA [8] that have explicit support for introducing a few custom instructions. Coupled with the fact that it is getting easier and cheaper for fabless companies to create their own cores and SoCs based on RISC-V, this has spawned a flurry of activity in identifying a few custom instructions for specific classes of applications. To start with, we conducted a very extensive study where we looked at 50+ papers that have been published in the last three years. We noticed that the flavour of current work [1], [2], [5], [9] in this area, is noticeably different. The two primary differences are as follows. ❶ Recent papers do not use automated techniques at all because of their nonspecificity and also due to the fact that they do not factor in design constraints; instead, the authors base their choices for custom instruction selection based on a combination of simple statistical analyses, intuitive reasoning, domain knowledge, and institutional memory. ❷ The second difference is that the aim is to keep at least 90% of the ISA the same, and only add a very few and carefully selected instructions to the ISA. ❸ We also observed that researchers take inspirations from other ISAs to add custom instructions in a new ISA to improve its performance. For example in references [1] and [2], researchers introduced custom instructions in the RISC-V ISA that were already available in the ARM ISA.

We need to understand that this trend is not unjustified. In general, any SoC vendor would not see a lot of benefit in creating a fully customised ISA – this will necessitate a new compiler tool chain. However, they would like to maybe have a very few new instructions that can lead to a disproportionate benefit in performance without unnecessarily complicating the design process. The celebrated example

in this space is the multiply-and-accumulate instruction in CNNs [5]. To identify such instructions, manual analysis with simple statistical reasoning is currently being used. In this paper, our utilitarian aim is to abet this process.

We propose a tool *ISAMod* that uses a novel method to visualise the execution of instructions in the program. Furthermore, it allows us to compare the executions of the same program when compiled with different ISAs and understand the differences in performance that arise from the ISA per se. We show using case studies that it is possible to identify instructions that can lead to a speed up. For instance, we show how we can use *ISAMod* to augment the RISC-V ISA and add a few custom instructions to disproportionately increase performance. The selling point of our tool is as follows. Designers can use it to augment their existing workflow that relies on basically institutional memory such that they find it much easier to analyse, understand, and assess the effect of custom instructions in program executions.

Our approach is inspired from similar tools [10] used in analyzing DNA fragments. They use visualization based methods to augment their extensive statistical analysis libraries to intuitively understand how cancer genomes differ from normal genomes. Figure 1 is a motivating example, where we compare the performance of different ISAs for the same architecture using the SPEC-2017 benchmarks. x86 is the best performer for 5 out of 6 benchmarks.

The contributions of our work are as follows:

❶ We compare three popular ISAs, x86, ARM and RISC-V, from a performance perspective using the SPEC-2017 benchmarks. ❷ We introduce two novel visualization techniques to quickly detect and understand the reasons for the differences in performance. ❸ Finally, we use the insights from the reasons to introduce custom instructions to the RISC-V ISA and improve its performance. We show that as compared to prior work, the benefits are similar; however, we reach the desired solution using a far easier and far more intuitive series of steps.

In Section II, we discuss the related work, discuss our simulation framework and the ISAMod tool in Sections III and IV, then proceed to Section V to discuss the evaluation results, and finally conclude in Section VI.

II. Related Work

Initial works [11], [12] in this area involved enumerating all possible subgraphs of a DFG for instruction identification. In the most general case, each node of the graph can either be included or excluded from a candidate instruction. Thus, the search space has an exponential complexity.

Later works [3], [13] provided algorithms for fast identification of custom instructions for ASIPs. It is done by applying different optimizations such as branch and bound techniques and imposing different constraints on the subgraphs (based on the number of inputs and outputs). Thus, these algorithms were much faster than the previous papers, but the time complexity was still exponential in the worst case.

Recent works [6], [7] try to scale the methods by applying distributed and parallel algorithms. They first partition the DFG into many smaller components and apply the enumeration algorithms on each partition in parallel. Hence, the effective run time of the enumeration algorithm has improved.

These works over-specialize the ASIP to work for only a particular benchmark. We propose a simple and intuitive tool in this paper that is not based on complicated DFG analyses; instead, we compliment the instruction selection process using different visualizations.

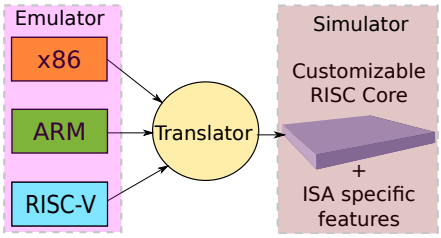


Fig. 2: Simulation framework

III. Simulation Framework

We want to find the effects of the ISA on performance for a given application. But the performance of a particular ISA is tightly coupled with its underlying micro-architecture if we use a real system. Other works have normalized the performance values based on the core frequency. But, there are many architectural parameters apart from the ISA on which the performance depends. Hence, to ensure a fair comparison, we use a simulation framework (see Figure 2) where we use the same RISC micro-architecture for conducting experiments for all the three ISAs. This method is commonly used in the research community to do such a comparative study [14], [15]. Since CISC ISAs such as x86 are internally converted to RISC ISAs [16], this method works well for both RISC and CISC ISAs and as a result it is the most popular approach in the architecture community for performing such kind of studies. However, to simulate some ISA specific artifacts such as shifted operands in ARM, it is necessary to introduce some special features in the simulator to support such special instructions. This has been done in our case in consultation with the processors' data sheets.

We compile all the code using the same version of gcc (8.2) with the same optimization flags and run them on the same version of the operating system (Linux 4.15) that was built with the same parameters.

We use the Tejas simulator [17] to do all our experiments. Tejas is a cycle-accurate architectural simulator and has been thoroughly validated against real hardware [17]. Our simulation framework is as follows. We simulate the different ISAs using a customizable RISC out-of-order pipeline core that simulates a Virtual Instruction Set Architecture (VISA), which is a RISC ISA. The instructions of the different ISAs are translated to VISA by a validated translation engine (for x86 we use PTLsim [16]). We simulate an out-of-order Intel-Haswell core (3.4 GHz core frequency), 32 KB L1-Data and Instruction cache, 256 KB L2 cache, and an 8 MB L3 cache.

We use QemuTrace, an extension of QEMU [18] to generate x86 and ARM instruction traces, and use Spike [19] to generate RISC-V instruction traces. Apart from instructions, we add branch taken/not-taken, and memory load/store address values in the trace. The Tejas simulator simulates these traces to compute cycles, energy, branch prediction rates, cache hit rates, and many more architectural statistics.

IV. The *ISAMod* Tool

The goal of this tool is to find the reasons for the differences in performance while running the same application on different ISAs in the same environment. The tool follows the steps shown in Figure 3.

1) Generate instruction traces for the same application for different ISAs.
2) Mark equivalent phases across the different traces by marking the start and end of a particular function call (the program counter of a function is obtained using *objdump*).

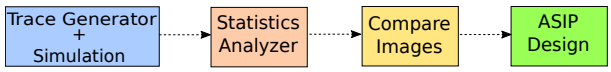


Fig. 3: The *ISAMod* tool

3) Simulate these traces using the Tejas simulator and compare the statistics across equivalent phases (each phase across different ISAs does exactly the same work).
4) Construct a pixelated image by mapping each instruction to a specific color.
5) Search for specific patterns in those images to understand the reasons for the differences in performance.
6) Use the patterns to improve the overall performance by creating an application-specific instruction set processor.

A. Statistics Analyzer

The input at this step is all the statistics generated using the Tejas simulator. We collect the executed instruction sequences, and performance figures for each phase across different ISAs. Along with this, we also collect the architectural parameters such as the number of LSQ forwarding events, branch prediction rates, and cache hit rates. The tool analyzes the most frequently executed functions and checks for performance differences across the different ISAs. Finally, it prints the sequences of instructions executed in these functions, which are sent to the image generator.

B. Image Generator

We create images for the most frequently executed functions using instruction traces for all the three ISAs. We start by mapping each micro-operation to a particular color and then create a 2D raster diagram representing the sequence of instructions horizontally from left to right (see Figure 4). We then define the following patterns for these images to help find the reasons for differences in performance.

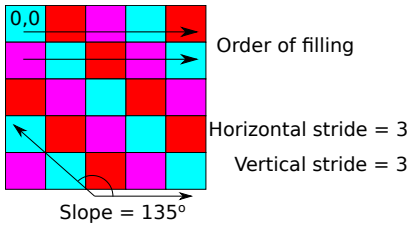


Fig. 4: Raster diagram from an instruction sequence

1) **Horizontal stride:** The horizontal stride represents the loop size. A smaller value is better.
2) **Vertical stride:** The vertical stride represents the loop count (number of iterations). It is relevant when we consider nested loops.
3) **Slope:** The slope of a particular color pattern can also be used to compute the loop size. See Figure 4 for the definition of the slope. In general, larger slopes correspond to a larger loop size.
4) **Color composition:** We have mapped darker colors to high latency operations such as loads, stores and floating point operations, whereas, we have mapped the low latency operations to lighter colors. Hence, it is very easy to visually identify which image is associated with a better performance.

For the current work, we do a manual analysis of the generated images. Using automated image processing and machine learning tools to automatically identify different features is part of future work.

C. Views

The *ISAMod* tool displays the comparison results in two views. In View I, we show the performance and architectural parameters using Circos plots, and the show the instruction sequences using a raster diagram in View II (refer Section V-B for a complete example). We used the Python imaging Library (PIL) to generate the raster diagrams and used the *circlize* [20] package in R to generate the View I plots.

V. Results

We compare the three ISAs using the SPEC-2017 benchmark suite. We do not use *x264* and *perlbench* since they have x86 ISA-specific code. RISC-V does not support cross-compilers for the Fortran programming language. Hence, we excluded those benchmarks. In Figure 5, we show the visualization of the most critical function (most frequently executed function) in all the three ISAs for a set of 6 benchmarks. Each image has 20×20 pixels: it displays a sequence of 400 instructions in a phase. Lastly note that we define **performance as the reciprocal of the simulated execution time.**

A. Results-I

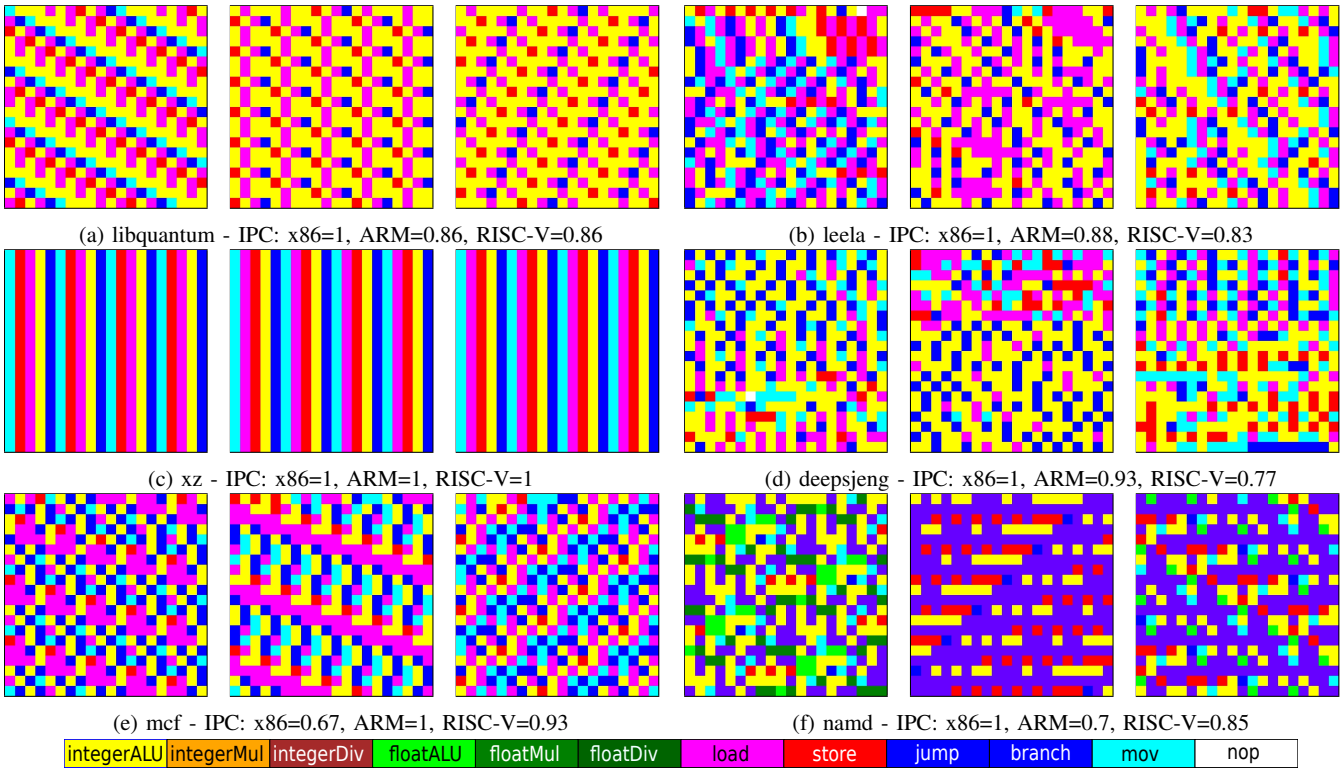
We primarily observe two different kinds of patterns. In Figures 5a, 5c and 5e, we observe a repeating pattern of colors representing a loop of instructions. In these benchmarks, the loop size varies from 5-30 instructions only. On the other hand, in Figures 5b, 5d and 5f, we do not find any simple repeating patterns. In these cases, we rely mostly on the color composition to understand the differences in the ISAs. Let us now discuss each benchmark in detail.

libquantum: It is a library for simulating a quantum computer and hence it is a computationally intensive benchmark. The most frequent function is *quantum_sigma_x*. We observed that x86 performs 14% better than the RISC ISAs. From Figure 5a, we observe that the horizontal strides of the three ISAs are 11, 12 and 13 respectively. A larger number of instructions are executed in a loop for the RISC ISAs as compared to x86 leading to poorer performance. The vertical stride is the same for all the ISAs, which means that the loop count is the same and we do not have nested loops. We also observed that the slope of the *blue* colored lines is the highest for the x86 architecture, hence pointing to a smaller loop size and better performance. Note that the x86 instructions are not complex instructions that require more cycles to execute. The composition of darker shades *blue*, *pink*, and *red* is the same for all the three ISAs. This correlates with a similar number of branch and cache accesses across all the three ISAs.

From further code analysis, we found that x86 performs better because it allows indirect memory accesses in arithmetic operations (see Figure 6). Hence, it reduces the total number of ALU operations (by 14%), leading to better performance.

xz: It is a compression library. We tested it using the *combined* input file. The *lzma_code* function is the most frequent function in this benchmark. The horizontal stride and the vertical stride are also the same across all the ISAs. The slope of the *blue* (branch) lines is also the same across all the ISAs (90^o). We also observed a similar color composition across all the ISAs (see Figure 5c).

From further code analysis, we observed that all the three ISAs run a similar set of five instructions (*mov, ld, st, alu and branch*) repeatedly. The only difference is that the order of loads and stores is interchanged in x86 as compared to RISC ISAs, which is insignificant in an OOO pipeline. Thus, we did not find any performance difference across ISAs for this benchmark.



(a) libquantum - IPC: x86=1, ARM=0.86, RISC-V=0.86

(b) leela - IPC: x86=1, ARM=0.88, RISC-V=0.83

(c) xz - IPC: x86=1, ARM=1, RISC-V=1

(d) deepsjeng - IPC: x86=1, ARM=0.93, RISC-V=0.77

(e) mcf - IPC: x86=0.67, ARM=1, RISC-V=0.93

(f) namd - IPC: x86=1, ARM=0.7, RISC-V=0.85

Fig. 5: Images for View II: x86 vs ARM vs RISC-V

mcf: It solves the vehicle scheduling problem in a public mass transportation system and is mostly comprises of integer arithmetic. The most frequently executed function is *primal_bea_mpp*. The function updates the costs of the individual objects using a *for* loop. The horizontal stride for the x86, ARM and RISC-V ISAs is similar (25, 23 and 28 respectively). From Figure 5e, we observe evenly spaced *pink* (loads) color lines in RISC-V, whereas they are closely packed in x86 code. The color composition of other colors is similar across the ISAs.

Since the loads are closely packed in the x86 ISA, this leads to a higher degree of memory and MSHR pressure translating to poorer performance. This explains the poor L1 and L2 cache hit rates in the case of the x86 ISA (25% and 43% respectively) as compared to RISC-V (43% and 56% respectively).

leela: It is an AI library for the Go playing engine that uses Monte Carlo simulation. The most frequently executed function is *FastBoard*. It consists of a series of *if-else* statements, hence we see a lot of conditional instructions (*blue* color in Figure 5b). The loop size is very large in this benchmark, thus we do not find repeating patterns in a window of 400 instructions. Hence, horizontal and vertical strides are not defined. But, we observe some features related to the color composition. We observe that many *mov* operations (*cyan*) in x86 are replaced with *ALU* operations (*yellow*) in ARM leading to a poorer performance in ARM (more structural hazards). A higher density of memory operations (*pink*, *red*) [20%] in the RISC ISAs as compared to x86 can also be attributed to the poorer performance: arises out of spilling because *mov* in x86 uses one less register.

deepsjeng: It is also an AI application for playing chess. The most important functions in this benchmark are *std_eval*, *qsearch* and *setup_attackers*. They call many smaller functions to define the moves of the individual chess pieces. Hence, this benchmark tests the function call-return efficiency of the different ISAs; this creates a lot of pressure on registers. Whenever, we have frequent functions calls, we need to add a lot of code to spill and load registers. Similar to *leela*, we do not observe highly repeating structures in this benchmark. But we observe differences in the color composition (see Figure 5d). The concentration of *pink* and *red* (load and store) colors is higher for ARM and RISC-V (16%) compared to x86. In such cases, an architecture that uses the stack might perform better because there will be more of load-store forwarding in the LSQ. Thus, x86 (uses the stack to store function arguments) has better performance as compared to ARM and RISC-V. The RISC-V ISA has a poor performance because it runs a larger number of micro-instructions (22% more compared to x86) for doing exactly the same work.

namd: It is a program for the simulation of large biomolecular systems. It is a computationally intensive application and consists of floating point operations. The key function in this benchmark is *calc_self_energy*. The *calc_self_energy* function consists of a sequence of load, sum-and-compare, and store operations. x86 has around 30% better performance compared to the the ARM ISA.

In Figure 5f, we observe a smaller number of *branch* operations (*blue*) in x86 compared to ARM and RISC-V, resulting in better performance. The composition of the memory operations (*pink* and *red*) is 20% lower in the case of x86. This is because x86 benefits from its complex conditional instructions , and its complex addressing modes.

B. Results-II: libquantum

In this section, we show the detailed output of the *ISAMod* tool for the *libquantum* benchmark. In Figure 6, we first show the performance and architectural parameters in View I using a Circos plot. For each concentric circular ring, the range of the radial axis

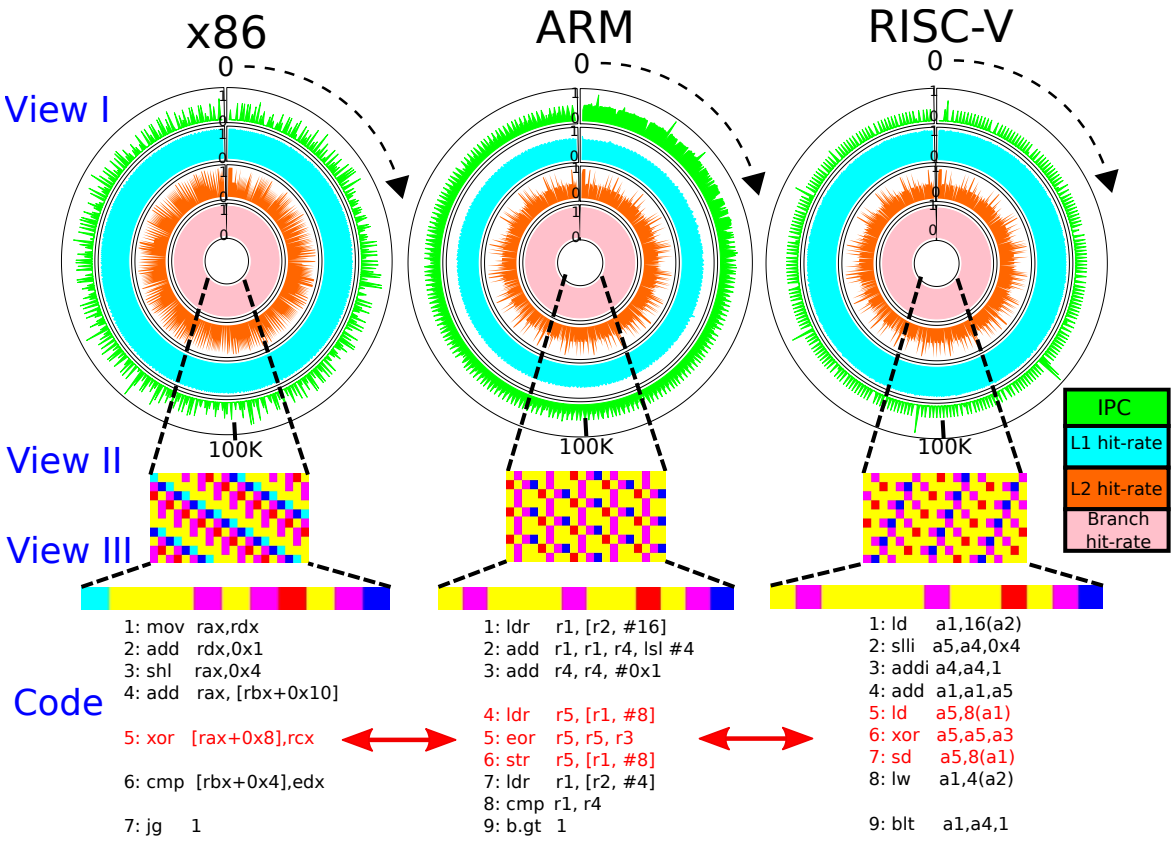


Fig. 6: libquantum: detailed analysis

is from 0 to 1. A point on a concentric ring of each plot shows the corresponding value for a set of 250 consecutive instructions. The complete circle corresponds to a total of 200K instructions (to be viewed in clockwise order starting from the top). We selected an instruction window of 200K instructions for the key function *quantum_sigma_x*. We observe higher IPC values in x86 as compared to the RISC ISAs. Similarly, the L1 and L2 hit-rates are higher in x86. The branch prediction hit-rate is almost equal to 100% for all the three ISAs.

In View II, we zoom into a 250-instruction phase and display the sequence of instructions executed in it. We observe higher horizontal stride values in the case of ARM and RISC-V as compared to x86. Next, in View 3, we zoom further and display the sequence of instructions corresponding to a single loop iteration. Along with the color map, we also display the sequence of assembly instructions of all the three ISAs.

In View III, we observe a larger number of ALU operations (yellow) in ARM and RISC-V as compared to x86. This can be verified from the code as we see a complex *xor* operation in x86 (line 5), which does a *load* as well as a *store*. The destination address (*rax+0x8*) is computed only once, and then the *load*, *xor* and *store* are performed. Whereas, in the case of RISC-V, this complex operation is split into three operations (Lines 5, 6 and 7). Here, the destination address (*a1 + 8*) is computed twice, once in Line 5 and again in Line 7. Rest of the instructions remain the same across the ISAs. The x86 ISA is benefiting from having a complex representation of the *xor* operation by minimizing the total number of address computations. This optimization is not applicable in the case of the RISC ISAs. In the case of RISC, we can save the extra ALU operation by using an extra temporary register but it will be adding to the register pressure. Thus, x86 has a 14% higher performance because of this optimization.

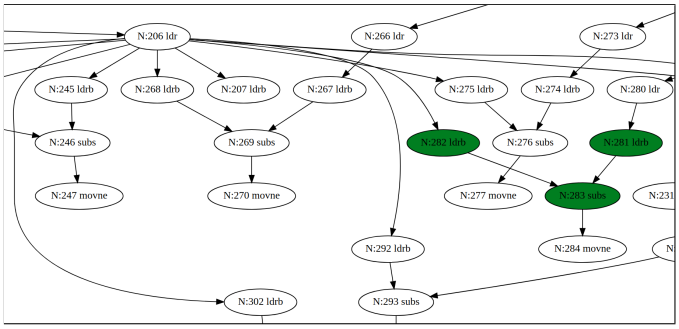


Fig. 7: Output of the FAGECI algorithm for *deepsjeng*

The *ISAMod* tool can in this way be used to analyze performance issues. The Circos plots help to find the relation between the performance and different architectural parameters. The visualization of the instructions helps us to easily identify the reasons for the differences in the performance. Finally, these reasons can be used to design an application specific instruction set processor (ASIP) as we show next.

C. Results-III: ASIP Design

In this section, we compare our results with the state of the art work in custom instruction generation, FAGECI [6]. We applied their algorithm of custom instruction enumeration and selection on the

TABLE I: Number of new custom instructions

Benchmark	FAGECI - Enumeration	FAGECI - Selection	*ISAMod*
xz	275	44	1
namd	301	53	3
libquantum	254	38	1
deepsjeng	215	31	8
leela	283	47	10

SPEC-2017 suite compiled for the RISC-V ISA. We set the number of inputs to 2 and the number of outputs to 1 for the custom instructions for their algorithm. First, we show a part of the DFG generated by their algorithm in Figure 7. The custom instruction subgraph is highlighted in green. This is the only visualization available in their work, whereas we provide three different views that are far more intuitive to select and analyze the custom instructions. Figure 7 is rather complex in nature and it is almost incomprehensible.

Now, we show the number of potential new instructions found using FAGECI in Table I. FAGECI enumerates 215 to 301 instructions and finally selects 31 to 53 new instructions. In comparison, *ISAMod* provides only 1 to 10 instructions, the output is thus much easier to analyze and understand.

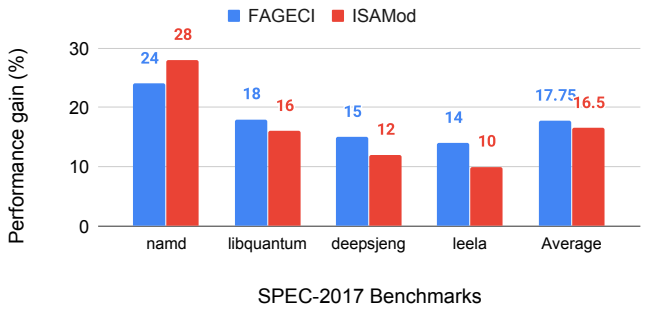


Fig. 8: Performance gain for RISC-V: FAGECI vs *ISAMod*

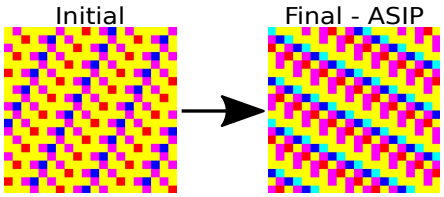


Fig. 9: libquantum - IPC: RISC-V=1 to RISC-V-ASIP=1.16

We show the improvements in the RISC-V core upon introduction of application specific instructions. For, the example of *libquantum*, we added a complex *xor* operation and did a simulation to get the new performance numbers. In Figure 8, we see that the performance in the case of *libquantum* increased by 16%. Figure 9 shows the raster diagram in the case of RISC-V before and after the addition of the *xor* instruction. We observe a reduction in the number of ALU operations graphically (reduced incidence of the *yellow* color). Similarly, in *leela and namd*, we observe a performance gain of 10% and 28% respectively after adding **just 2-10 new instructions**. The average performance gain by incorporating the new instructions identified by *ISAMod* is **16.5%**. We also incorporated the new instructions provided by the FAGECI algorithm, and the average performance gain is **17.75%** for the SPEC benchmarks. The results are comparable. Our algorithm's benefit is that we provide fewer new instructions and a visualization engine to analyze the new instructions and explain the performance differences.

VI. Conclusion

Instead of relying on costly graph analyses, in this paper we relied on a novel method that uses graphical techniques. Our set of increasingly complex diagrams filter out all unnecessary details and show us all the information that is required to identify the key reasons behind the performance differences between frequently executed functions of the same benchmark compiled with three different popular ISAs. The Circlos plots capture micro-architectural details and the raster diagrams capture code patterns. We show that we can find the reasons by correlating these diagrams with small assembly code snippets that are executed repeatedly. We made use of this information to increase the performance by 10-28% of basic RISC ISAs such as RISC-V by adding 2-10 new instructions. The performance gain is comparable with the state of the art in custom instruction generation; we at the same time reduce the number of infructuous and trivial suggestions made by custom ISA generators by 5-40 times.

References

[1] M. Theiley, V. Hoang, and Y. Yarom, "Risc-v isa custom extensions for use in cryptography," in *ICR*, 2019, p. 58.

[2] S. Payvar, M. Khan, R. Stahl, D. Mueller-Gritschneder, and J. Boutellier, "Neural network-based vehicle image classification for iot devices," in *SiPS*. IEEE, 2019, pp. 148–153.

[3] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *CASES*, 2004, pp. 69–78.

[4] P. Bonzini and L. Pozzi, "A retargetable framework for automated discovery of custom instructions," in *IEEE ASAP*, 2007, pp. 334–341.

[5] R. Porter, S. Morgan, and M. Biglari-Abhari, "Extending a soft-core risc-v processor to accelerate cnn inference," in *CSCI*, 2019, p. 694.

[6] H. Chen and S. Chen, "Fast automatic generation of efficient custom instructions for application-aware computing," in *ICACI*. IEEE, 2018, pp. 283–288.

[7] S. Wang, C. Xiao, W. Liu, and E. Casseau, "A comparison of heuristic algorithms for custom instruction selection," *Microprocessors and Microsystems*, vol. 45, pp. 176–186, 2016.

[8] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[9] M. S. Louis *et al.*, "Towards deep learning using tensorflow lite on risc-v," in *CARRV*, 2019.

[10] L. Choy *et al.*, "Constitutive notch3 signaling promotes the growth of basal breast cancers," *Cancer research*, vol. 77, no. 6, p. 1439, 2017.

[11] P. Faraboschi *et al.*, "Lx: a technology platform for customizable vliw embedded processing," in *ISCA*, 2000, pp. 203–213.

[12] M. Arnold and H. Corporaal, "Designing domain-specific processors," in *CODES*, 2001, pp. 61–66.

[13] N. T. Clark, H. Zhong, and S. A. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1258–1270, 2005.

[14] A. Venkat and D. M. Tullsen, "Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor," in *ISCA*. IEEE, 2014, pp. 121–132.

[15] M. DeVuyst, A. Venkat, and D. M. Tullsen, "Execution migration in a heterogeneous-isa chip multiprocessor," in *ASPLOS*, 2012, pp. 261–272.

[16] M. T. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator," in *ISPASS*. IEEE, 2007, pp. 23–34.

[17] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter, "Tejas: A java based versatile micro-architectural simulator," in *PATMOS*, 2015.

[18] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.

[19] "Spike: Risc-v isa simulator," https://github.com/riscv/riscv-isa-sim, accessed: 2020-09-01.

[20] Z. Gu *et al.*, "circlize implements and enhances circular visualization in r," *Bioinformatics*, vol. 30, no. 19, pp. 2811–2812, 2014.