Empower: Automated Architecture Specialization with RISC-V

***Abstract*—Compute demand continues to grow in the post-Moore era leading to a resurgence of specialized architectures that achieve energy-efficient computing. Across the compute spectrum from embedded low-power sensing devices to high-performance virtualization datacenters, hardware specialization has emerged as an important knob in meeting the need for scalable, energy-efficient architectures. While prior research has addressed the performance and energy benefits of fixed-function hardware accelerators, the opportunities with specializing a general-purpose Instruction Set Architecture (ISA) have not been as widely investigated. The popular open customizable *RISC-V* architecture presents a unique opportunity to explore specialization of general-purpose ISAs.**

In this work, we present *Empower* - a framework for automated specialization of instruction-set architectures. Starting with a baseline *RISC-V* RV32IMC configuration, *Empower* automatically identifies new instructions given a set of input benchmarks/applications to be accelerated. Breaking program execution into sub-blocks, *Empower* conducts compiler-like dataflow analysis to identify candidate new instructions. The framework automatically names, and encodes new instructions into the custom opcode space provisioned by the *RISC-V* ISA specification. On applications from the Embench benchmark suite and CoreMark, *Empower* improves performance by as much as 40%.

***Index Terms*—Instruction Set Architecture, Design Automation, Specialization, Customization**

I. Introduction

Continued demand for high performance compute capability coupled with the end of Moore's Law has led to a surge of interest in domain-specific acceleration techniques [2]. From tiny ultra low-power sensors to high-performance data center servers, leveraging application characteristics to build specialized hardware enables both energy and performance benefits [3], [4]. While specialized hardware brings energy-efficiency, it comes at a cost: it is hard-wired to accomplish a very specific function and could be completely unusable when newer computational paradigms have to be realized. Thus, general-purpose programmable architectures such as Intel x86, ARM Cortex-A, ARM Cortex-M and RISC-V continue to remain dominant with offloading some functions to dedicated hardware. In particular, specialization is an attractive design alternative in low-power embedded systems where instruction fetch-decode energy is often a significant portion of overall energy consumption.

In order to address inefficiencies with traditional general purpose Instruction Set Architectures (ISAs), these architectures have incorporated specialization in different ways. For instance, the ARM Cortex-M33 supports custom instructions [5]: a semiconductor chip maker could add their own specialized instructions and execution units (combinatorial logic) to the baseline ISA supplied by ARM. Similarly, RISC-V offers a variety of standard extensions and custom extensions to allow tailoring the ISA to different application segments.

Specializing the ISA takes work. Typically, experts in the application domain work with computer architects, designers and compiler developers to identify new instructions. A prototype model of the proposed new instructions must be developed to explore the changes. This requires a careful analysis of how to transform code written for the baseline ISA to leverage new instructions. Finally, the specification, design, and compilation tools must all be updated to incorporate new instructions. Specialized tools and flows have been developed in academia as well as industry to improve the efficiency of this task of ISA specialization. However, while current open-source and/or industry tools reduce the burden of *implementing* new instructions, they do not address the problem of *finding* the right instructions.

Given a mix of applications drawn from an application space, our tool – *Empower* – *automatically* recommends new instructions that help accelerate these applications. *Empower* incorporates compiler-like data-flow analysis to identify patterns of computations (or sequences of instructions) that could be transformed into new instructions. These transformations are performed keeping RISC principles and design feasibility in mind. A goal of *Empower* is to allow architects to control the nature and types of instructions emitted, such as, for example, specifying whether 3- source operand instructions are allowed or not. *Empower* provides a fast ISA design-space exploration capability that aids expert-driven definition. It also estimates potential savings in execution cycles if identified new instructions were implemented, thereby allowing quantitative architectural decision-making.

In this work, we make the following original contributions:

1) We present *Empower*: a tool that fully automates the process of identifying new specialized instructions over the RISC RV32IMC [9] baseline.
2) We evaluate the tool using Embench and Coremark benchmark suites and find that the tool improves performance by as much as 40%.
3) We analyze the feasibility of implementing the tool-discovered instructions.
4) We present an illustration of the capability of our tool by exploring new instruction opportunities when the traditional RISC 2-operand rule is relaxed.
5) Our tool is an open-source Python tool, promoting further development and innovation within the RISC-V ecosystem.

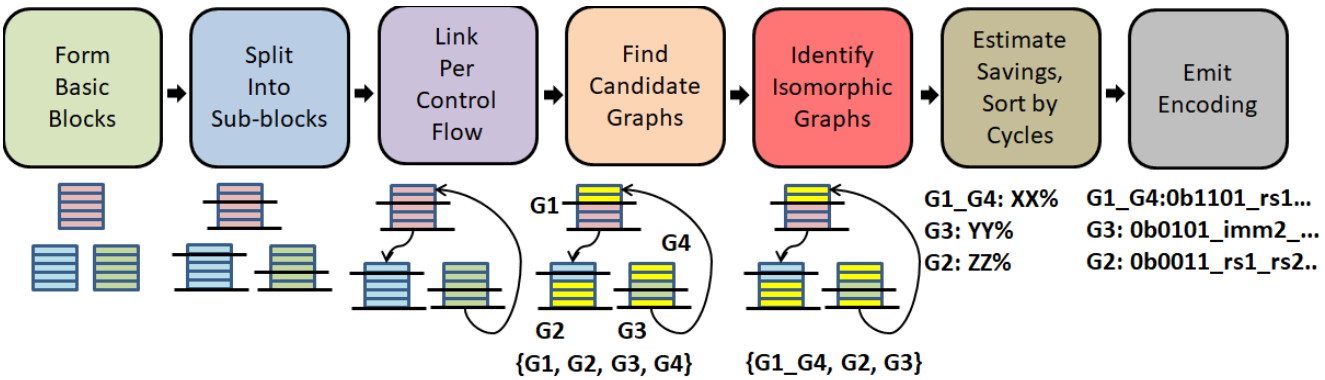

Fig. 1. *Empower* Workflow

II. *Empower* Overview and Usage

Empower is a trace-driven tool implemented in Python. Application execution traces (sequence of PCs and corresponding instructions) are supplied as input to the tool along with a set of options to control the specialization tasks. The tool parses input traces and constructs an internal representation of program control and data flow. In order to do this, *Empower* is equipped with an ISA decoder: instruction encodings and semantics of instructions from the RISC-V RV32IMC architecture configuration[1] are decoded to extract various pieces of information. This includes register sources and destinations of ALU instructions, control transfers and branch destination PCs, range of immediate constants encountered, and so on. For example, if the trace contains the instruction word "0x97ba", the tool decodes this as the compressed add instruction *c.add a5, a5, a4* that adds contents of registers $a4$ and $a5$ and saves the result in $a5$. *Empower* is thus able to use traces of execution that list the instructions executed along with their execution frequencies. Each entry in the trace is a tuple consisting of *PC*, *instruction word*, *execution frequency*. For example, the tuple $(0x1014e, 0x97ba, 1024)$ denotes that the instruction word $0x97ba$ at PC $0x1014e$ executes 1024 times. From this entry, *Empower* uses its ISA decoder to extract the information that the instruction at this PC reads registers $a4$ and $a5$ and outputs register $a5$. The execution frequencies are recorded against basic blocks that the tool extracts (described in Section III).

Empower performs compiler-like program control flow analysis and data-flow analysis on the input traces to obtain data-flow graphs that are inter-linked through program control. Finally, *Empower* identifies computational sub-graphs that could be transformed into specialized instructions adhering to the following requirements:

1) ***2R1W:*** The specialized instruction shall adhere to the RISC rule of two source operands and one destination operand, where all operands come from registers (the *2R1W* rule). Under control of a user-specified option, the two-source rule could be relaxed. We explore this in Section V.
2) ***Depth:*** Depending on microarchitecture and semiconductor technology used, how much "work" an individual instruction is allowed to do varies. This is determined by the clock timing critical path in the design. The addition of a complex instruction that performs a lot of work in a single cycle may violate clock timing for the intended clock speed. Thus, *Empower* can be configured to limit its specialization only to graphs with a limited depth (e.g., depth not exceeding 4 compute "steps") or to graphs that contain only certain sub-sets of instructions (e.g., do not specialize graphs that contain floating-point instructions).
3) ***Liveness:*** Replacing a pattern of computation with a single new RISC instruction in the context of overall program execution requires preserving the outputs that the original computation produces. If an intermediate instruction in the graph produced a register output that was expected to be "live" at the end of that graph, then it would be functionally incorrect to reduce the computation to a single instruction that eliminates the intermediate register update. Thus *Empower* carries out "liveness" analysis [7] to ensure that the new instruction can correctly replace the computational graph *in situ* without side-effects.

We describe the detailed design of *Empower* in the next section.

III. *Empower* Design

Figure 1 outlines the workflow of *Empower*. We describe each step below:

Form Basic Blocks: The tool processes execution traces to construct basic blocks. It uses the standard "leader" algorithm [6] to track leader PCs where new basic blocks start and iterates until no new leaders are found.

Split into Sub-Blocks: As the tool does not track memory addresses, currently, it takes a conservative approach of splitting basic blocks into sub-blocks that are delineated by memory accesses [2]. This allows the tool to analyze short sequences

[1]In the future, we plan to incorporate additional RISC-V extensions that are standardized.

[2]In the future, we plan to relax this restriction.

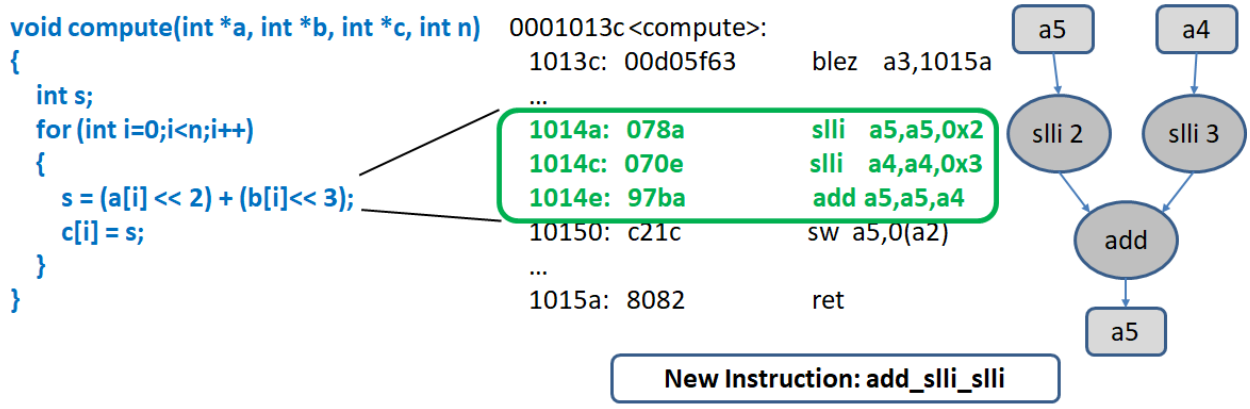

Fig. 2. Example of *Empower* Discovering a New Instruction

of ALU instructions (possibly representing a sub-expression) without any intervening memory instructions. This decision has one major advantage: the search for new instructions becomes very efficient as the scope is restricted to sub-blocks.

Link Per Control Flow: Since new instructions must preserve "liveness" guarantees, *Empower* links all the basic blocks corresponding to an execution trace into a single whole program-wide graph. This is performed using control transfer instructions encountered in the trace.

Find Candidates: This is the most computationally intensive step where the tool first constructs several candidate graphs for each sub-block and then verifies if the candidates meet the requirements of *2R1W*, Depth and Liveness described in Section II.

A candidate graph is constructed starting with every PC in each sub-block and recursively adding edges to preceding PCs where the PCs exhibit a consumer-producer relationship based on which registers are they reading/writing. If no prior PC in that sub-block produces a register value, then the register value is considered "live" upon entry into the sub-block.

The *2R1W* rule is verified by tracking all the register operands that are read by instructions in the graph. Graphs that need more than two source operands are discarded.

The *Depth* rule is easily verified by computing the depth of the acyclic graph and discarding graphs that are greater than the specified depth. Similarly, if the graph comprises instructions that are disallowed by the user, the graph is discarded.

Verifying the *Liveness* rule [7] requires tracking if any of the intermediate instructions' outputs are live beyond the scope of the graph. That is, if an instruction in a successor sub-block requires this intermediate computation value, then reducing the graph into a single instruction will violate the *liveness* guarantee and therefore this transformation can not be performed.

Formally, let $live(Bx)$ denote the set of registers that are needed to be live on entry into a block (or sub-block) Bx. Let $use(Bx)$ denote the set of registers that are used (read) by Bx (before perhaps being written to); $define(Bx)$ denotes registers written to inside Bx. Now, if Bc denotes a successor of Bx, then we define $live(Bx)$ as:

$live(Bx) = use(Bx) \cup (\bigcup_{Bc} live(Bc) - define(Bx))$

Thus, *Empower* computes $use(B), define(B)$, and $live(B)$ for all sub-blocks. Once these sets are computed, *Empower* checks if any intermediate register write inside a candidate graph G in a sub-block Bx is in the live-set of any successor sub-block Bc. Mathematically, if $def(I(G))$ denotes an intermediate instruction's output, then:

$invalid(G) =\mid \{r \mid \exists_{Bc} \exists_{I(G)} r \in def(I(G)) \cap live(Bc) \mid > 0$

$invalid(G)$ is true if the size of such a set is non-zero, i.e, at least one intermediate register write exists that needs to be live in some successor. Additionally, if an intermediate write is used in any other graph within the same sub-block, again the transformation is similarly invalid. For simplicity, we omit mathematically describing this check but it is a straightforward extension of the above.

It may be observed that our analysis is conservative. Whole program-wide analysis would potentially reveal additional opportunities for specialization. We defer this for future work.

Graphs that pass all three checks go to the next step.

Perform Isomorphism Checks: In our experiments, we observed that identical computational patterns often repeat across different programs or even in different parts of the same program. Thus, the tool gathers all such identical patterns into a single new instruction by performing subgraph isomorphism checks across the candidates. Note that we compare only the opcodes of the instructions in the sub-graphs and ignore actual registers used and any differences in values of immediate constants.

We reduce the cost of this step by encoding each sub-graph as a string such that if two strings hash to the same bucket only then a more detailed isomorphism check is carried out.

Estimate Cycles Saved: For each new instruction, the tool computes the total savings achieved by it. In our model, every instruction (baseline or new) executes in one cycle. Thus, if the new instruction subsumes N baseline instructions, then it saves $(N-1)$ cycles each time it executes. Since the tool processes execution traces, it is able to provide an exact measure of cycles saved under this model. In Section V we explore this single-cycle assumption in more detail.

Emit New Instructions: Finally, the tool emits new in-

structions. This includes the instruction name, the instruction opcode fields, register operands, and immediate operands. Instruction-naming is automatically performed by traversing the corresponding computational graph and concatenating the constituent instructions' names.

A. *Example of* Empower *at work*

In this section, we illustrate the operation of *Empower* with a simple example outlined in Figure 2. On the left, we show the original C code that was used: a function named *compute* that iterates through two input arrays, performing shifts and addition of array elements to an output array. The middle column shows the assembly view of the code produced by the compiler[3]. A trace of execution of this code is analyzed by *Empower* to recommend new instructions. Among all the sub-blocks that the tool identifies, we have highlighted the sub-block of interest that holds three instructions starting at PC 0x1014a. The first two instructions perform shifts and the third adds the shifted operands. Taken together, these three instructions satisfy the requirements of *2R1W* (only two registers $a4$ and $a5$ supply source operands to this computation), *Depth* (the depth of this graph is only two) and *Liveness* (no subsequent instruction relies on the intermediate value produced in register $a4$). Thus the computation graph (shown in the right-hand column) qualifies as a candidate new instruction and is recommended by the tool, along with an estimate of its potential savings. In this example, since three instructions are subsumed by the proposed instruction, it saves two cycles in each invocation.

IV. Experimental Evaluation

We evaluate *Empower* using traces obtained from the Embench [1] and CoreMark [10] benchmark suites. We chose these benchmark suites as they are widely used for embedded microcontroller performance evaluation and have a diversity of codes. These benchmark codes were compiled for RV32 *IMC* (I=Integer baseline, M=Multiplication, C=Compressed). Since *Empower* does not yet support other extensions, we exclude benchmarks that make extensive use of floating-point operations. Traces of execution are obtained by executing these codes on the RISC-V SPIKE [11] simulator.

For design timing, we use the open-source *ibex* [8] 32-bit RISC-V core implementation as our baseline. We incorporate new instructions to this baseline and synthesize the core using the Yosys [12] synthesis tool with the FreePDK 45nm design library [13]. We verified that the baseline core synthesizes at 250MHz CPU clock speed.

V. Results

Figure 3 presents total cycles saved as a percentage of total (original) execution cycles across the Embench suite of benchmarks as well as the Coremark benchmark. Gains vary significantly depending on the benchmark with *aha-mont64* showing negligible improvement while *matmult-int* showing nearly 40% reduction in cycles. On average, *Empower* has achieved over 7% reduction. We emphasize that this improvement has been achieved entirely automatically using conservative analysis. In embedded systems where issues such as code size and instruction fetch energy are important considerations, this reduction enables valuable savings.

[3]For brevity, we have not shown all of the instructions emitted by the compiler.

A. *Top Five Instructions*

While *Empower* performs a full search of the program and identifies several new instructions, interestingly, the vast majority of savings come from very few new instructions. Figure 5 depicts this for a few benchmarks. For each benchmark, it plots cumulative cycles saved by new instructions, where the instructions are organized in decreasing order of their savings. In each plot, the top 5–10 instructions yield the biggest benefits as can be seen by the steep slope at the beginning and a near-flat slope subsequently. Thus, subsequent instructions contribute a negligible amount to savings.

Figure 4 plots the savings achieved by only the top five new instructions in each benchmark as a fraction of the total savings achieved by all of the new instructions for the benchmark. Closer the bar is to 1.0, the greater the savings achieved by the top five. Barring *picojpeg*, in all benchmarks, the top five achieve over 80% of savings. Benchmarks such as *matmult-int* and *nettle-aes* are particularly interesting in that the topmost instruction contributes to over 90% possible savings. For example, Figure 6 visualizes the top-scoring instruction identified by *Empower*. It is of depth 3 and meets timing.

This result is encouraging: it demonstrates that *Empower* produces practical new instructions that can be incorporated into design without adding a large instruction decode overhead.

B. *Instruction Complexity*

We evaluate the instruction recommendations for their complexity. By *complexity*, we refer to the depth of the instruction's computation graph. Larger the depth, the more complex the instruction. Thus, from a design timing perspective, it is preferable to incorporate instructions with a smaller depth. Figure 7 shows a breakdown of savings achieved by instruction depth. Across all the benchmarks, on average more than 90% savings come from instructions with depth ≤ 4. Thus, *Empower* is combining short sequences of baseline computations into new instructions.

C. *Impact of Timing*

Very complex instructions may not meet the single-cycle execution criterion. In order to verify if *Empower* produced such instructions, we implemented complex instructions into the *ibex* core [8] and verified if clock timing was violated. Using the open-source *yosys* design synthesis tool and the 45nm freePDSK design library, the baseline synthesizes at 250MHz. While the vast majority of new instructions met this timing, a few instructions that incorporate multiplication/division functionality from the "M" extension of the baseline violated timing by about 4%. In such cases, the micro-architecture may choose to implement a multi-cycle instruction to avoid slowing down the clock. A detailed analysis of such micro-architectural changes is not in the current scope of this work.

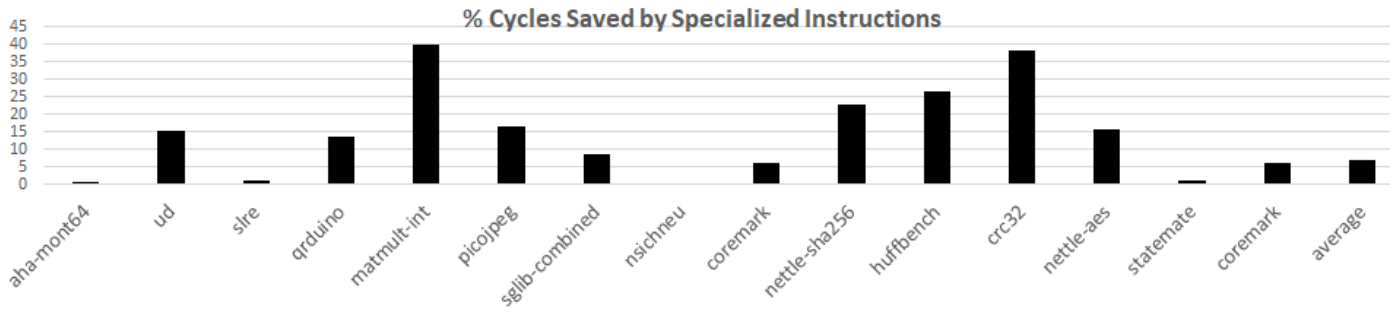

Fig. 3. Performance Improvement with *Empower*

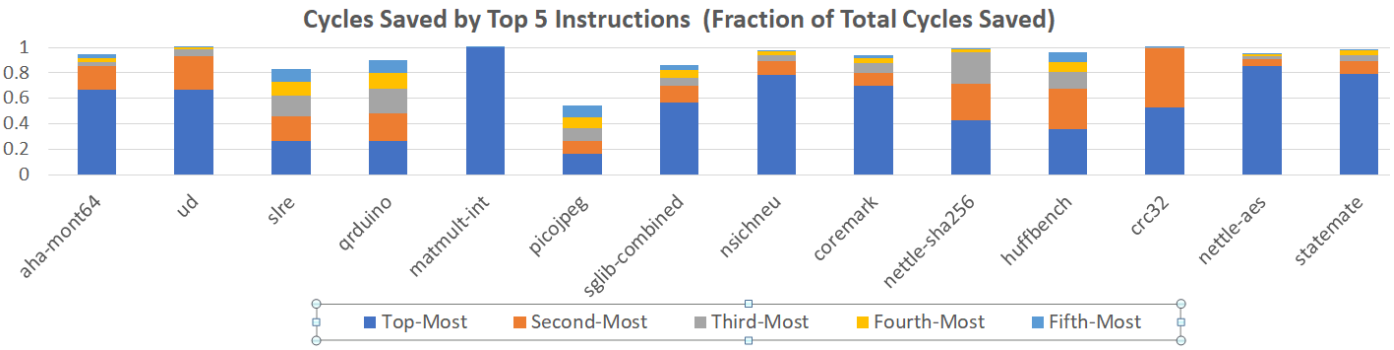

Fig. 4. Improvement with Top-5 Specialized Instructions

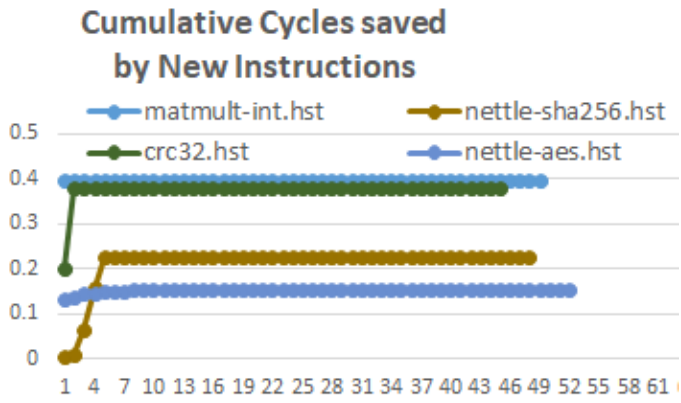

Fig. 5. Cumulative Cycles Saved by New Instructions

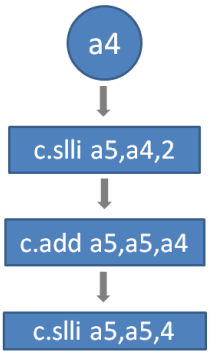

Fig. 6. Top-Scoring Instruction in *matmult-int* Benchmark

D. *Relaxing the* 2R1W *Rule*

A benefit of our tool-driven automated approach is the ability to explore various changes to the ISA. In this section, we evaluate the benefit of relaxing the RISC two-source operand rule. In our experiment, we allow upto three source operands per instruction (*3R1W*). Figure 8 plots the additional savings achieved by *3R1W* over the baseline *2R1W* rule for a subset of benchmarks.

The relatively moderate gains suggest that three-operand instructions should be carefully considered for inclusion, given the cost of the additional read port on the register file.

VI. Related Works

Specialization and instruction generation has received significant attention in both academia and industry. Here, we briefly compare *Empower* with some relevant prior works.

Several EDA tools have been developed to automate the generation of digital design and software tools using a machine-readable ISA specification. LISATek [14] is a language and tool that generates both design files as well as an instruction-set simulator and compiler for the specified ISA. Tensilica [15] provides technology to accelerate the design of signal-processing oriented processor designs. In the context of RISC-V, Codasip [16] and Imperas [18] provide software solutions to profile application code and to design and verify new instructions. Unlike these commercial endeavors, *Empower* goes a step further: it automatically identifies new candidate instructions. Further, *Empower* is an open-source tool meant to promote further development and innovation in the RISC-V ecosystem.

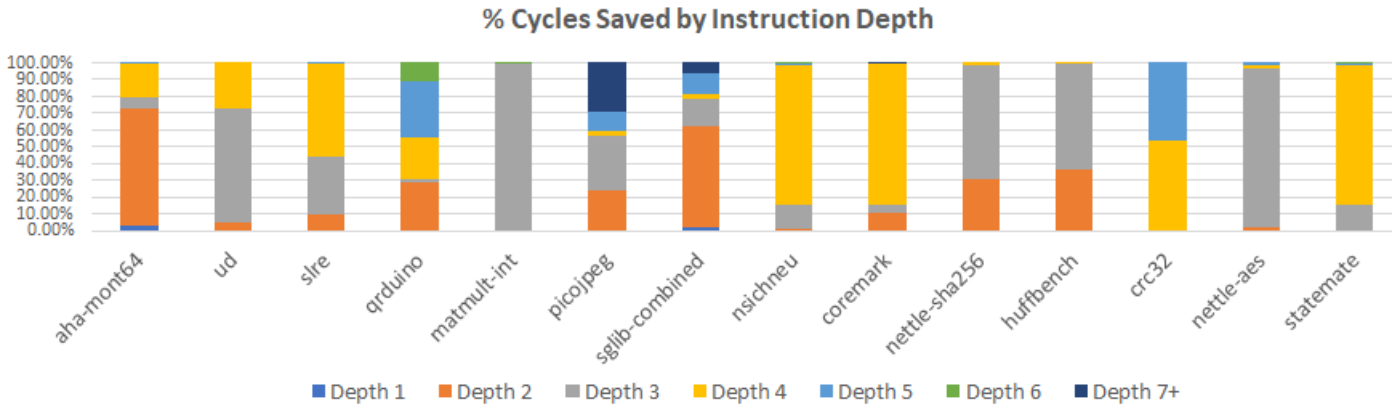

Fig. 7. Breakdown by Instruction Depth

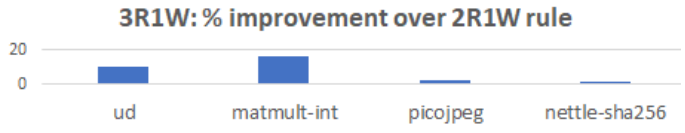

Fig. 8. Relative Improvement with *3R1W* over *2R1W*

The work in [18] is conceptually similar to ours but they target a specific Custom Computing Accelerator (CCA) [19]. In contrast, our work does not assume a specific accelerator mapping or technology and enables architects to explore potential savings using different microarchitectural back-ends. The CCA work [19] proposes an architectural template for offloading computations. Our work is orthogonal to this and *Empower* can be used as a front-end atop CCA. The work in [20] deals with the issue of the large search space for finding candidates and proposes algorithms to speed-up the search. As our work breaks the analysis down to sub-blocks, we avoid the problem of a very large search space.

VII. Conclusion

In this paper, we presented *Empower* – an automated framework for specializing the RISC-V instruction-set architecture. Using a representative suite of benchmarks, we demonstrated that *Empower* can discover useful new instructions automatically, thereby reducing the cycle-time for exploring application-specific customizations. On some benchmarks, *Empower* achieves as much as 40% savings. Further, being a free & open-source Python-based tool, it is easily extensible to support other architectures and more sophisticated analysis capabilities by contributors in the RISC-V ecosystem. In the future, we plan to extend *Empower* to identify new instructions that span basic blocks, exploit sub-word packed arithmetic, or present vectorizable opportunities. Finally, we plan to systematically evaluate opportunities when the *2R1W* principle is relaxed.

References

[1] FOSSi Foundation,"Embench^{TM}: A Modern Embedded Benchmark Suite," embench.org

[2] W. J. Dally, Y. Turakhia, S. Han, "Domain-specific hardware accelerators," Communications of the ACM, July 2020.

[3] R Hameed et al, "Understanding sources of ineffciency in general-purpose chips," Communications of the ACM, 2011.

[4] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," ISSCC, 2014.

[5] ARM, "ARM Custom Instructions," www.arm.com, 2019.

[6] GeeksForGeeks, "Basic Blocks in Compiler Design," https://www.geeksforgeeks.org/basic-blocks-in-compiler-design/, 2020.

[7] Wikipedia, "Live variable analysis," https://en.wikipedia.org/wiki/Live_variable_analysis, 2020.

[8] A. Bradbury et al, "Ibex RISC-V Core," https://github.com/lowRISC/ibex, 2020.

[9] RISC-V Foundation, "RISC-V Processor Architecture," www.riscv.org, 2020.

[10] EEMBC, "CoreMark - An EEMBC Benchmark," https://www.eembc.org/coremark/, 2020.

[11] A. Waterman et al, "Spike RISC-V ISA Simulator," https://github.com/riscv/riscv-isa-sim, 2020.

[12] Claire Wolf, "Yosys Open Synthesis Suite," http://www.clifford.at/yosys/, 2020.

[13] NC State, "FreePDK3D45TM," https://research.ece.ncsu.edu/eda/freepdk/freepdk45/, 2011.

[14] R. Muhammad, L. Apvrille, R Pacalet, "Evaluation of ASIPs Design with LISATek," https://link.springer.com/chapter/10.1007/978-3-540-70550-5_20, 2008.

[15] Cadence, "Tensilica Customizable Processor and DSP IP,", https://ip.cadence.com/ipportfolio/tensilica-ip, 2020.

[16] Imperas, "Methodology for Implementation of Custom Instructions in RISC-V Architecture,", https://riscv.org/wp-content/uploads/2019/02/Imperas-EW-2019-Custom-Instructions-booth-slides-KM.pdf, 2020.

[17] Codasip, "Codasip Studio," https://codasip.com/tag/customization/, 2020.

[18] P. Bonzini, L. Pozzi, "A Retargetable Framework for Automated Discovery of Custom Instructions," International Conference on Application Specific Systems (ASAP), Architectures and Processors, 2007.

[19] N. Clark, A. Hormati, S. Mahlke, "Scalable subgraph mapping for acyclic computation accelerators," Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, 2006.

[20] P. Yu, T. Mitra, "Scalable custom instructions identification for instruction set extensible processors," Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, 2004.