Virtual Prototype driven Design, Implementation and Evaluation of RISC-V Instruction Set Extensions

Milan Funck[1] Vladimir Herdt[1,2] Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
Milan.Funck@dfki.de, vherdt@uni-bremen.de, drechsler@uni-bremen.de

***Abstract*—RISC-V is a modern open source *Instruction Set Architecture* (ISA) and designed in a very extendable manner, which allows for highly application specific solutions. However, the identification of suitable instruction set extensions usually requires a significant manual effort and therefore is a very challenging process. In this paper we propose a lightweight alternative methodology to find suitable application-specific RISC-V extensions, using a *Virtual Prototype* (VP). This is done, purely by observing the used instructions during the execution of the targeted application on the VP. Therefore, no further information about the application itself are needed. In this context the advantages and the flexibility of a VP and the straightforward extendability of the RISC-V ISA are demonstrated.**

I. Introduction

Development of application specific integrated circuits is an expensive and tedious process, but leads to way more efficient solutions then conventional multipurpose-processors [1]. Due to its extendability, the RISC-V *Instruction Set Architecture* (ISA) serves the gap between these two approaches. RISC-V originated in 2010 at the University of California, Berkeley, with the intend to introduce a new and well thought through open source ISA, serving as a stepping-stone for further development in this area. The first official description was released in 2011 and the RISC-V foundation was founded in 2015 [2]. RISC-V offers a mandatory base integer instruction set in combination with a set of optional standard instruction set extensions. Moreover, RISC-V is designed from the ground up to support integration of additional custom instruction set extensions. This provides everything needed, for a efficient application specific processor design and a manageable development process. Except for the compliance with the RISC-V instruction format, there are in principle no restrictions to the functionality of the custom instructions and therefore the main challenge lies in the design of the instruction set extensions itself. Finding such an extension often requires multiple extensive analyses combined with complex graph theory and is very application dependent [3], [4].

In this paper we instead propose a *Virtual Prototype* (VP) driven approach, which works for any application and is tailored specifically for the RISC-V ISA. By observing the executed RISC-V instructions at run time in the Instruction Set Simulator (ISS) of the VP, potential optimizations are identified. The application itself is treated as a black box and the goal is to find an appropriate sequence of instructions, that could be replaced by a custom instruction in accordance with the RISC-V ISA, resulting in an increased efficiency of the application. By using VPs, our analysis can be performed in the early stages of the design process, hence it is suitable for early evaluations of RISC-V instruction set extensions. The identified instruction set extensions can also be analyzed and tested directly afterwards at the VP level. We use the open source RISC-V VP, available at GitHub [5], as foundation to build our VP-driven analysis approach. The VP is implemented in the standardized SystemC language using the industrial proven *Transaction Level Modeling* (TLM) style [6], [7]. Our experiments demonstrate, that our VP-driven approach is suitable to efficiently identify and test application specific instruction set extensions for RISC-V.

The paper is structured as follows: following a discussion on related work (Section II) and relevant background information (Section III), we will start with discussing the theoretical background and the requirements for such a VP-based analysis tool (Section IV). Then, we discuss an implementation and integration of such a VP-driven analysis and show evaluation results based on a case study (Section V-A). Finally, we conclude the paper with a discussion on future work (Section VI).

II. Related Work

Analysis and integration of application specific instruction set extensions is an important research topic. [8] investigate a compiler based approach to analyze basic blocks to identify instruction extensions. [9] developed an approach to find candidate instructions for speeding up neural network applications. In [10] an end-to-end flow is investigated which supports in implementing instruction set extensions based on the GCC tool chain. To support the implementation process of custom instructions, [11] investigate compiler techniques to provide support for new instructions effectively. Other approaches that identify custom instruction set extensions for example rely on a feedback-driven approach to obtain candidates [3] or analyze hardware accelerator blocks to provide a tightly coupled instruction set extension with less communication overhead [4].

For RISC-V specifically, we are not aware of academic work targeting identification of general purpose application specific instruction set extensions. Nonetheless, there exists work on designing use-case specific instruction set extensions for example for cryptographic operations [12]. In addition, the steadily growing RISC-V ecosystem offers tools to describe and implement instruction set extensions. Noteworthy approaches are based on the CoreDSL [13] and SAIL [14] descriptions. They enable to specify the functionality of the RISC-V instruction set extension at a high-level of abstraction. Such description language are complementary to our approach as they can help in specifying implementing the instruction set extension after it has been identified by our approach. With regard to RISC-V, several instruction set extensions already exists which are standardized or in the process of being standardized. Such new instruction set extensions are being specifically designed to cover a large set of applications, for example the instruction set extensions for multiply, vector or bit operations. The work [15]

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project VE-HEP under contract no. 16KIS1342 and within the project VerSys under contract no. 01IW19001.

978-1-6654-9431-1/22/$31.00 ©2022 IEEE

reports results on enhancing the RISC-V ISA in order to improve code density by modifying the available instructions in the standard instruction sets themselves. Finally, there exists proprietary tools, such as provided by Codasip, to integrate custom instruction set extensions in order to build domain specific processors [16]. At the hardware level, another recent research direction proposed to leverage a combination of FPGAs with a RISC-V processor to enable reconfigurable instruction sets at run time [17].

III. Preliminaries

This section reviews relevant background information on RISC-V (Section III-A) as well asl SystemC TLM and the RISC-V VP (Section III-B).

A. RISC-V ISA

The latest version of the RISC-V ISA includes a base integer instruction set for 32 and 64 bit architectures and a first draft for a 128 bit version. In addition to this, the RISC-V ISA contains several standard extensions, such as the floating point extension, the multiplication extension or the compressed instruction extension. Also, there is a number of op-codes reserved for custom instruction set extensions [18]. Every instruction follows a certain instruction format, with the 7 lowest bits being the main part of the instructions opcode. There are essentially six types of instruction, depending on the constellation of used registers and immediate values. For every instruction that uses one or more of the source and destination registers, their position in the instruction is always the same. The rest of the available bits is used to either encode an immediate-value or extend the opcode space. Except for the 7-bit opcode, custom instructions however do not necessarily have to follow this convention and the remaining bits can be used to encode whatever is desired and required for application specific purposes.

B. SystemC TLM 2.0 and RISC-V VP

SystemC [6] was introduced by the Open SystemC Initiative (OSCI) in the year 1999. It is an extension of the C++ class library and enables the modeling of hardware behavior or rather the modeling of time, reactivity, hierarchy and concurrency. By using communication mechanisms and hardware data types an event based simulation core can simulate a desired hardware model. The RISC-V VP uses a TLM 2.0 bus to connect the individual modules via memory mapping. This way, data can be exchanged between the modules from the software-side simply just by performing read- and write-operations to the respective memory-address. A transaction-object is then used inside of the VP to simulate the data-transmissions. Therefore, only a TLM-interface and a certain unused address-space is needed to add a new module through the TLM-ports. Finally, a basic timing model is incorporated into the RISC-V VP, which can be modified in the ISS to update the RISC-V instruction execution timings and used in conjunction with the TLM 2.0 timing capabilities to suit the individual application [19].

IV. Dynamic Instruction Analysis

In this section we describe our proposed approach for a dynamic analysis method to design and evaluate application-specific instruction set extensions. First, we start with an overview of the core concept, which is aided by a short example and then followed by more detailed descriptions for the general procedure.

The goal of this approach is to find combinations of executed instructions that could be replaced by a simple custom instruction, resulting in less needed processor-cycles and instruction memory space. For example the ADDI-instruction adds a given immediate-value to the value of a given register and stores the result in a given destination-register. This instruction could be used to increment a loop-iterator. In this case only one register is used and the immediate-value would probably be 1. Hence the bits used to encode two of the three parameters are wasted. If say a jump-instruction to the beginning of the loop would follow, these bits could be used to encode the offset-value for the jump.

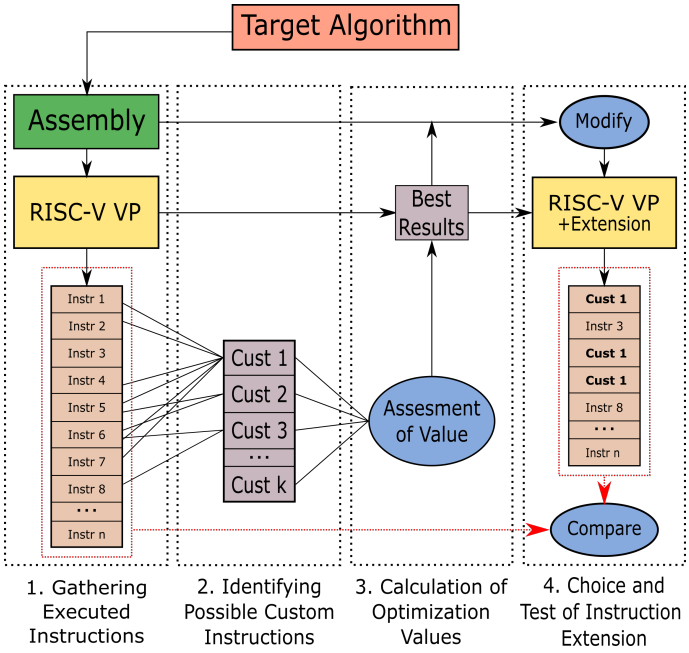

Fig. 1. Overview on general procedure

This example shows, that certain combinations of instructions could be encoded in a single one, if they are reduced to their situational purpose. If the exact same sequence of instructions is used very often during the execution of a certain algorithm, it could be worth it, to use a custom extension instead. Therefore, mainly two questions have to be answered, in order to formulate an application-specific optimization proposal:

- Which executed instructions could be combined into a new instruction?
- What added value would this new instruction actually bring?

To answer these questions, we designed a VP-based dynamic instruction analysis approach, that operates sequentially in four main steps. Fig. 1 shows an overview of these four steps, which will be described in the following respective subsections.

A. Gathering Executed Instructions

The first step is to gather a sample of executed instructions representative of the application. The target algorithm should be run inside of a simulation, close to the real target application environment and in a representative scenario. The results of the analysis will depend mostly on the quantity of sub-sequences of executed instructions in relation to the overall executed instructions. Hence it is advised to limit the number of gathered instructions to where these proportions stay fairly stable. Apart from the gathered instructions it is beneficial to store some additional information about them, that could be useful for analysis. This includes the number of executed instructions, the number of executed cycles for each instruction and in total, which extensions are used, how often each of the single instructions is executed and how many different parameters are used.

B. Identifying Possible Custom Instructions

In the second step potential candidates for an instruction set extension are identified. Each of them is represented by at least one sub-sequence of the representative sequence of executed instructions and a corresponding custom instruction, that is able to replace it. A certain sequence of instructions can only be replaced by a custom instruction, if the execution of the latter results in the same outcome, as the original sequence would. From the software-side the only limitation of a custom instruction set extension is the number of parameters (registers and immediate-values), that can be encoded in the given instruction length. The actual execution of the corresponding operations has then to be implemented in hardware. The Example in the beginning of this section shows, that some parameters of executed instruction are not always needed, for example when the source-register and the destination-register are the same. Therefore the main objective is to identify every sub-sequence of executed instructions, that operates on a set of parameters, which could be encoded in a single instruction. The following strategies and considerations can be used to do so and will be explained using the exemplary assembler code seen in Fig. 2:

1) Collecting Registers and Immediate Values: If the same register or the same immediate value is used by more then one executed instruction of a sequence, it has to be encoded just once. For example in Fig. 2 the first two instructions use a total of two different registers and both use the same immediate value. Therefore the first custom instruction displayed in Fig. 2 could replace them.

2) Constant Immediate Values: Most immediate values use a large amount of the available encoding space. However they do not necessarily have to be encoded and instead the used immediate value can be considered as constant for the custom extension. For example in Fig. 2 the second and the third instruction use a total of two different registers and two different immediate values. By choosing one of them as constant the second custom instruction displayed in Fig. 2 could replace them.

3) Constant Zero-Register: The RISC-V ISA specifies the zero-register as a constant (hardwired) 0. Therefore it does not necessarily have to be encoded, because using it as a destination-register has no effect and using it as a source-register can be replaced by using the value 0 directly. For example in Fig. 2 the fourth instruction is used to subtract the value of register x3 from the value of register x0 and store the result in the register x4. The actual outcome is x4 being the negation of x3 and therefore encoding x0 would not be necessary in a corresponding custom instruction.

4) Skipping Instructions: When sequentially trying to fit the parameters of multiple executed instructions in to a single one, there obviously comes the point, where the next one does not fit anymore. Under certain circumstances the next executed instruction can be skipped to still make use of the remaining encoding space. Of course, the skipped instruction then has to be executed after the resulting custom instruction in the modified application code and this is only possible, as long as the resulting change in the execution order does not change the overall outcome. Effectively that means, that an instruction can only be skipped and executed later, if it does not write to registers, that would have been read from afterwards and if it does not read from registers, that now are written to, before the delayed execution. For example in Fig. 2 the third and the fourth instructions use a total of four different registers and one immediate value, which would be to many parameters to fit into a custom instructions. However, changing the execution order by swapping the fourth and the fifth instruction would have no effect on the outcome, meanwhile the third and the fifth instruction could actually be replaced by a custom instruction, because they only use

```
1 ADDI x3, x1, 15   ->  x3 = x1 + 15
2 ADDI x1, x1, 15   ->  x1 = x1 + 15
3 ADDI x2, x2, 11   ->  x2 = x2 + 11
4 SUB x4, x0, x3    ->  x4 = x0 - x3
5 ADD x2, x1, x2    ->  x2 = x1 + x2
6 ADDI x4, x4, 9    ->  x4 = x4 + 9
7 ADDI x5, x5, 11   ->  x5 = x5 + 11
```

```
1 cust1 r1, r2, X   ->  r1 = r2 + X, r2 = r2 + X
2 cust2 r1, r2, X   ->  r1 = r1 + X, r2 = r2 + 11
```

Fig. 2. Exemplary Assembler Code

a total of two different registers and one immediate value, which can be encoded in the available RISC-V custom instruction set format.

5) Multiple Solutions: For obvious reasons there can be multiple different possible custom instructions which could replace a certain sequence of executed instructions. For example as already established above, the second and the third instruction in Fig. 2 could be replaced with a custom instruction, by choosing at least one of the immediate values as constant. The same goes for the sixth and the seventh instruction, but when choosing the second immediate value as constant, which in both cases is eleven, the corresponding custom extension could actually replace both sub-sequences. In fact, the whole exemplary sequence of instructions shown in Fig. 2 could be replaced by a single custom instruction, when all the mentioned strategies are used. The downside would be, that the corresponding custom instruction would only be able to replace a very specific sequence of instructions. Hence the resulting increase of efficiency of the resulting instruction set extension would depend on the target application. These examples show that it is best, to consider the corresponding custom extensions for every possible combination of instructions and constant/encoded parameters and the decision of which ones are suited best for the target application should be completely left to the third step, which will be explained in the next subsection.

C. Calculation/Estimation of Optimization Values

The third step takes all the possible custom extensions identified before, which then have to be evaluated and compared to find the ones best suitable to optimize the target application-scenario. For this reason the effective value of each of them has to be calculated or estimated. This value is a product of the number of times, the respective sub-sequence can be replaced in the representative sequence of executed instructions (due to overlapping, this might not be the number of times, the sub-sequence was actually found) and the effective benefit from replacing this instruction string with a corresponding custom instruction during execution. This benefit is dependent on the chosen optimization goal and on the technical feasibility. For example when aiming for a higher execution speed, the number of saved processor cycles during execution could be used, but it would be necessary to calculate the number of execution cycles each of the identified custom instructions would save, when replacing/parallelizing the corresponding sequence of instructions, thus being highly dependent on the actual hardware implementation. To simplify this step it is helpful to make some assumptions about the shortest path of multiple instructions and estimate this value. When implemented, the evaluation and the presentation of results has to be highly parameterized and extendable, to allow for detailed adjustments according to a specific optimization goal.

D. Choice and Test of Instruction Set Extensions

In this last step an optimization proposal is formulated. One or more custom instructions are chosen from the established results,

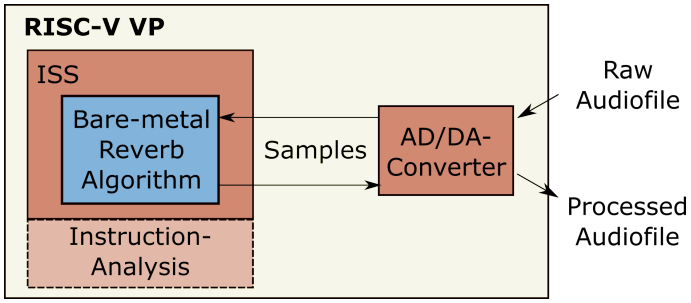

Fig. 3. VP-Architecture for the Case-Study

implemented and tested. If this choice is done manually, it is advised to limit and sort the established results on base of certain parameters, because the list of possible custom instructions could potentially be very long. As soon as a choice was made, a simulation based implementation of the instruction set extension is the obvious starting point. The application code and/or the used compiler have to be modified to use the new custom instructions instead of the respective sequence of instructions. Adaptable simulation approaches like the RISC-V VP would allow for simulating and testing interactions with the application environment rather precisely, thereby evaluating the analysis results and the optimization proposal.

V. Evaluation

We have implemented our proposed approach using the open source RISC-V VP as a foundation. In this section we present evaluation results using a reverb algorithm as a case study. In the following we analyze the exemplary target application and the influence of the different analysis parameters as well as the process of adding instruction set extensions on VP-level is examined. Furthermore the analysis results of the reverb algorithm are then compared to those of a number of benchmark-algorithms, giving a first impression of what to expect from this method (Section V-D). The VP's main components for the case-study are displayed in Fig. 3 and will be explained in the following sections. Afterwards the analysis results will be summarized and discussed (Section V-E).

A. Case Study - Reverb Algorithm

As the application in the center of the case study serves a reverb algorithm. It takes a stream of audio samples as an input and applies a number of different delays to it, before outputting a desired mix of the original and the processed audio stream. The result is a dense layer of echoes creating an impression of space to the audio signal and a device like this is commonly used in audio production environments. The actual implementation is based on the Schroeder-Reverberator [20] and the algorithm is written in C/C++, using the standard cross-compiler contained in the RISC-V toolchain [21]. Because of the fact, that the resulting algorithm almost entirely consists of two different types of delays, that are applied over and over again, a high potential for optimization is to be expected. The goal is, to run this algorithm on a 32-bit RISC-V processor and design a simple cost efficient instruction set extension containing up to two custom instructions, to increase the execution efficiency. In a real-world scenario the samples would be provided at a certain sample rate by an AD-/DA-converter generating them from the incoming analog audio signals. Furthermore it would convert the processed samples back to the analog domain and pass them on to the output of the overall device. Therefore a module is added to the VP as (see Fig. 3), simulating this exact process. To do so, a given exemplary wave-file is read in, extracting the samples and sequentially providing them through memory mapping (TLM). A bare-metal version of the reverb algorithm is run on the core, accessing and manipulating these samples.

B. Implementation of the Analysis-Tool

For this case-study, the analysis-tool is integrated into a 32-bit version of the RISC-V VP. Every instruction that is decoded by the ISS is also copied into a list inside of the analysis-class. Once the simulation of the target application is finished, said list is sequentially run through and starting from each instruction, the analysis-tool tries to fit as many consecutive instructions into the given instruction space. This is done by using the strategies mentioned in section IV-B. For every possible variation of constant and encoded parameters, that would suffice to encode a subsequence of instructions, a new custom instruction is created. Each unique custom instruction as well as the respective number of equivalent instances found (minus the ones that would overlap) is then added to a second list and the individual optimization values are calculated, according to the chosen analysis parameters. For Example, in order to calculate the number of cycles, that a custom instruction would need to execute a sequence of instructions all at once, certain assumptions about the shortest path were made for this case-study. Especially reading/writing from/to memory was excluded from possible parallel execution. The analysis-class contains a number of parameters, the first of which being the maximum number of instructions to be gathered. This can be used to reduce the complexity and execution time of the analysis for longer and/or repetitive applications. The next block of parameters influences the identification of the possible custom instructions. This includes the maximum number of instructions a custom instruction would replace, the maximum number of constant immediate values, that can be used and the maximum number of instructions that can be skipped, if possible. Finally a number of parameters can be used to determine how the resulting list of found instruction strings should be evaluated and how detailed the results together with the recommended instruction set extensions should be displayed.

C. Simulation and Verification of the Suggested Instruction Set Extensions

The instruction set extensions suggested by the analysis tool are tested as described in section IV-D. This requires them to be included into the simulation process. Names and opcodes have to be registered in the RISC-V VP, so the ISS is able to decode and execute them. To simplify the addition of different custom extensions, the actual execution is handled by an external class, which uses a certain interface and implements the desired functionality and its timing. For any custom instruction to be actually used, whether in a simulation or otherwise, the instructions also have to be registered in the RISC-V compiler. Specifically the opcodes and the corresponding bit-masks, defining the structure of the instructions as well as the instruction definitions specifying the number/kind of encoded registers and immediate values have to be added. Once the respective files contain these information, the custom instructions can be used via inline-assembly or in the assembler code directly. The application C++ file is compiled to assembler code and modified by manually replacing the instruction sequence in question with the new custom instructions. This new assembler file is then compiled and executed on the RISC-V VP with the corresponding instruction set extension enabled. For the case-study the analysis revealed, that the following two custom instructions together would bring the highest increase in efficiency:

| **I-Immediate** | **r1** | **empty (3 Bit)** | **r2** | **0001011** |

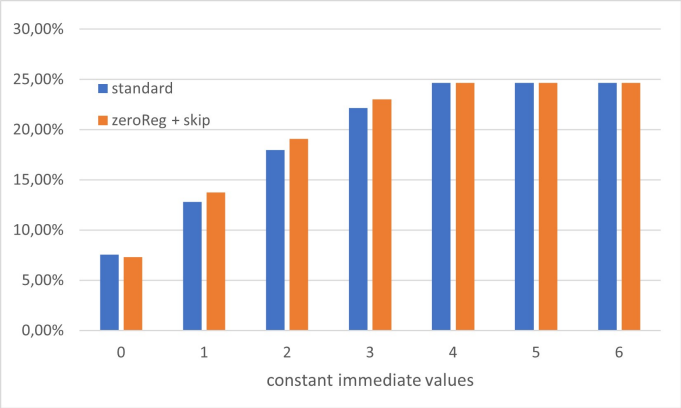

Fig. 4. Average percentage of instructions saved with one instruction over all measured algorithms

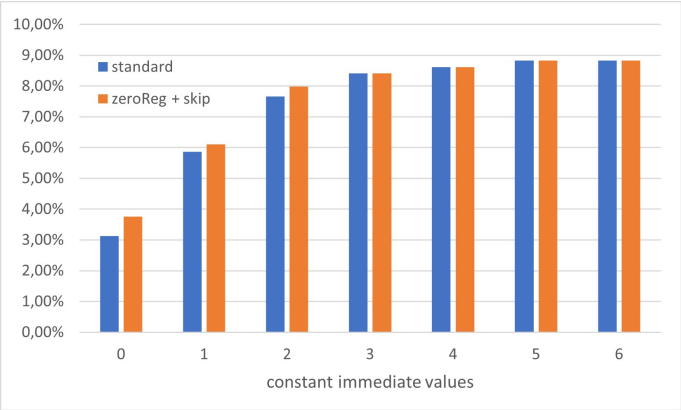

Fig. 5. Average percentage of cycles saved with one instruction over all measured algorithms

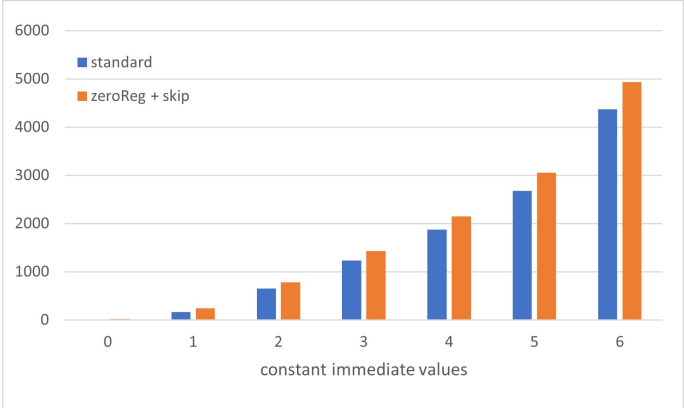

Fig. 6. Average number of possible custom instructions identified over all measured algorithms

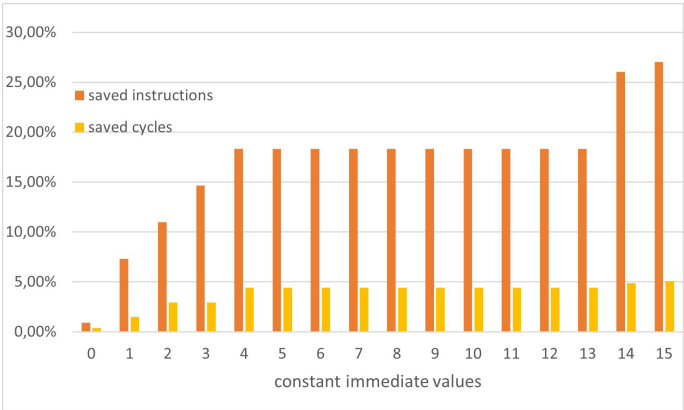

Fig. 7. Fibonacci-Algorithm: percentage of cycles saved with one custom instruction

Replaced Instructions:
ADDI r1, r1 Immediate
SLLI r2 , r2 , 2 (constant)
ADD r2 , r1, r2

| **I-Immediate** | **r1** | **empty (8 Bit)** | **0101011** |

Replaced Instructions:
LUI r1, 73728 (constant)
LW r1, r1, Immediate

Still it is worth mentioning, that the modification of the assembler code might not always be as trivial as replacing a certain sequence of instructions. In case of the primes benchmark algorithm for example, the custom instruction suggested by the analysis-tool is supposed to replace a sequence of instructions starting with a conditional jump. In other words the particular sequence observed during execution only happens, whenever the jump condition is fulfilled. This leaves the question, how the resulting custom instruction should be implemented and how its hardware equivalent would actually behave.

D. Analysis Results

The influence of the different analysis parameters is examined by performing the analysis on the different algorithms with varying configurations and comparing the outcome. The analysis of the reverb algorithm shows, that the number of constant immediate values available has a significantly higher influence on the quality of the suggested optimizations, than the addition of a constant zero register or the skipping of instructions. The same holds for the other benchmark algorithms. For instance Fig. 6 displays the average number of different possible custom instructions identified for the algorithms, in relation to the analysis parameters. The exponential correlation to the number of constants is clearly visible and the time needed to finish an analysis run rises in a similar fashion.

Fig. 4 and Fig. 5 show the average instructions and cycles that could be saved for the algorithms, when using the corresponding best two custom instruction derived from the analysis results. With a maximum of four to five constant immediate values, the optimization potential does not increase any further. Only an extended analysis of the Fibonacci-algorithm with up to 15 constant immediate values reveals, that at some point almost a fourth of the entire algorithm could be encoded in one single custom instruction. In Fig. 7 it can be observed, that this still does not lead to any real improvements in terms of saved processor cycles.

E. Discussion

The case study shows, that above all the use of constant immediate values enables a compression of multiple instructions into one. A maximum number of at least five constant immediate values should be considered, to reach the best potential optimization values. Beyond that the increase of constants does not lead to any actual improvements, even though replacing instruction sequences of extreme length might be possible. In case of the Fibonacci benchmark algorithm almost the entire executed instruction sequence could be fitted into a custom instruction. Obviously the

TABLE I
IDENTIFIED OPTIMIZATION POTENTIAL

Algorithm	saved instructions	saved cycles	savable cycles
Reverb	25,39 %	16,01 %	30,82 %
Fibonacci	32,98 %	8,86 %	20,42 %
Qsort	17,82 %	4,21 %	15,18 %
sha512	18,58 %	5,92 %	26,18 %
Primes	49,36 %	11,57 %	34,06 %
Mergesort	8,95 %	3,13 %	19,5 %

feasibility and the use of such an instructions is to be questioned. All of this can be explained with the simple fact, that encoding the immediate values takes up the most of the instruction space and taking them out of the equation leaves more space for the registers. In a majority of cases the compiler performs a number of operations on a group of registers, that is small enough to be encoded in a single instruction and therefore it can replace the whole corresponding instruction sequence. Treating the zero-register as constant can occasionally contribute to this effect. The approach of skipping instructions on the other hand, might reach certain optimization values without as many constants, but overall does not create better results. There is no need to skip an instruction, as soon as there is enough encoding space to include it into the string. The fact that all these observations are based on the used compiler and the algorithms chosen, has to be taken into account. Also the algorithm dependent optimization potential in conjunction with the chosen timing model and optimization premises is no direct indicator for the quality of the optimization approach itself. For that reason, the results of the analysis-tool should be compared to the maximum potential optimization possible. Table I shows the percentage of saved instructions and saved cycles, when using the best combination of two custom instructions suggested by the analysis tool for the respective algorithm. Additionally the last column contains the maximum percentage of executed cycles that could be parallelized, following the optimization premises used for the analysis. To summarize, the analysis-tool implementing the proposed approach delivers promising results. For the reverb algorithm from the case study, a sequence of instructions was identified, whose replacement with a custom instruction could save up to 15,27% of the executed instructions and up to 9,63% of the processor cycles needed. In combination with a second one, 25,39% of the executed instructions and 16,01% of the processor cycles needed, which is fairly impressive.

VI. CONCLUSION AND FUTURE WORK

The extendability of the RISC-V ISA proves to be very useful. The base instruction set is similar to common architectures and with the addition of a custom extension, a RISC-V processor can meet most individual demands. The possibilities of extension are almost endless, especially on the software-side. With only a few modifications, the already established RISC-V compilers can compile the new instructions via inline-assembly or by modifying the assembler code directly. In the end, the feasibility of the hardware implementation is the only limiting factor. Using a RISC-V VP, as shown in this paper, is crucial to finding, simulating and verifying such an extension. Because of its adaptable structure it can be tailored to suit any specific situation. The use of a dynamic instruction analysis-tool like the one proposed in this paper shows very promising results. Of course, the potential for optimization is very application depended, though the reverb-algorithm used for the case study turns out to be a worthy candidate. The analysis-tool was able to identify a possible increase in efficiency around 16%, using only two simple custom instructions, that should not be to complicated to implement in hardware. The overall idea of replacing strings of instructions, observed during executing is simple and effective. Especially the use of constant immediate values facilitates fitting multiple instructions into one.

Regarding the handling of jump operations and the general estimation of the implementation effort on both the hard- and software side, there is still room for improvement. Combining the dynamic instruction analysis with a static analysis of the application code and an automated generation of the modified assembler code could be helpful additions. Another strategy to consider is to save on encoding some of the registers, similar to the use of constant immediate values. Alternatively it could be allowed for one or more custom instruction to have internal registers if necessary. Apart from these improvements, an approach like this could be used to analyze a wide range of different applications and combine the results, in order to find further potent extensions. The application itself is treated like a black box and almost no background knowledge about it is required. Like the simulation itself, the analysis-tool can be adjusted to fit the goals and restrictions of a desired optimization.

REFERENCES

[1] C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 2, may 2011. [Online]. Available: https://doi.org/10.1145/1968502.1968509

[2] (2020) RISC-V website. [Online]. Available: https://riscv.org

[3] U. Kastens, D. K. Le, A. Slowik, and M. Thies, "Feedback driven instruction-set extension," *SIGPLAN Not.*, vol. 39, no. 7, p. 126–135, jun 2004. [Online]. Available: https://doi.org/10.1145/998300.997182

[4] N. Clark, W. Tang, and S. Mahlke, "Automatically generating custom instruction set extensions," 2002.

[5] "RISC-V virtual prototype," https://github.com/agra-uni-bremen/riscv-vp.

[6] *IEEE Standard SystemC Language Reference Manual*, 2011.

[7] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.

[8] D. Shapiro, M. Montcalm, and M. Bolic, "Parallel instruction set extension identification," in *2010 IEEE 26-th Convention of Electrical and Electronics Engineers in Israel*, 12 2010, pp. 000 535 – 000 539.

[9] D. Shapiro, J. Parri, J.-M. Desmarais, V. Groza, and M. Bolic, "Asips for artificial neural networks," *SACI 2011 - 6th IEEE International Symposium on Applied Computational Intelligence and Informatics, Proceedings*, 05 2011.

[10] O. Almer, R. Bennett, I. Böhm, A. Murray, X. Qu, M. Zuluaga, B. Franke, and N. Topham, "An end-to-end design flow for automated instruction set extension and complex instruction selection based on gcc," *1st International Workshop on GCC Research Opportunities, Paphos, Cyprus*, 04 2012.

[11] A. Murray and B. Franke, "Compiling for automatically generated instruction set extensions," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 13–22. [Online]. Available: https://doi.org/10.1145/2259016.2259019

[12] B. Marshall, G. R. Newell, D. Page, M.-J. O. Saarinen, and C. Wolf, "The design of scalar aes instruction set extensions for RISC-V," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 1, p. 109–136, Dec. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8729

[13] "Coredsl-instruction-set-description of RISC-V," https://github.com/Minres/RISCV_ISA_CoreDSL.

[14] "Riscv sail model," https://github.com/riscv/sail-riscv.

[15] P. Petrov, "Enhancing the RISC-V instruction set architecture," 2019.

[16] Z. Přikryl, "Creating domain-specific processors using custom RISC-V ISA instructions," 2020.

[17] N. Dao, A. Attwood, B. Healy, and D. Koch, "Flexbex: A RISC-V with a reconfigurable instruction extension," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 190–195.

[18] (2021) RISC-V isa manual git-repository. [Online]. Available: https://github.com/riscv/riscv-isa-manual

[19] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *Journal of Systems Architecture*, vol. 109, p. 101756, 02 2020.

[20] M. R. Schroeder, "Natural sounding artificial reverberation," Journal Of The Audio Engineering Society, 1962.

[21] (2020) RISC-V toolchain - git repository. [Online]. Available: https://github.com/riscv/riscv-gnu-toolchain