RVVRadar: A Framework for Supporting the Programmer in Vectorization for RISC-V

Lucas Klemmer
Institute for Complex Systems
Johannes Kepler University
Linz, Austria
lucas.klemmer@jku.at

Manfred Schlägl
Institute for Complex Systems
Johannes Kepler University
Linz, Austria
manfred.schlaegl@gmx.at

Daniel Große
Institute for Complex Systems
Johannes Kepler University
Linz, Austria
daniel.grosse@jku.at

ABSTRACT

In this paper, we present RVVRadar, a framework to support the programmer over the four major steps of development, verification, measurement, and evaluation during the vectorization process of an algorithm. We demonstrate the advantages of RVVRadar for vectorization on several practical relevant algorithms. This includes in particular the widely-used libpng library where we vectorized all filter computations resulting in speedups of up to 5.43. We made RVVRadar as well as all benchmarks (including the RVV-based libpng) open source.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; • **Hardware** → *Hardware accelerators*; • **Information systems** → *Open source software.*

KEYWORDS

SIMD, Vector, RISC-V, RVV, Vectorization, Software Optimization

ACM Reference Format:
Lucas Klemmer, Manfred Schlägl, and Daniel Große. 2022. RVVRadar: A Framework for Supporting the Programmer in Vectorization for RISC-V. In *Proceedings of the Great Lakes Symposium on VLSI 2022 (GLSVLSI '22), June 6–8, 2022, Irvine, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3526241.3530388

1 INTRODUCTION

Accelerating the execution of software has always been a major goal. If the software at hand exhibits *Data Level Parallelism* (DLP) like for instance matrix-oriented computations or media-oriented image processing, *Single Instruction Multiple Data* (SIMD) extensions [5] allow significant performance gains. SIMD achieves this by executing the same operations on multiple data elements (a vector of data). The SIMD concept has already been explored in the 1970s, e.g. popularized in the supercomputers by Cray, and these early approaches have become known as *vector architectures* [4]. Since roughly 20 years, general purpose processors feature SIMD instructions which fall into the category of *multimedia extensions* [9]. Prominent examples include MMX, SSE, and AVX from Intel, or NEON from ARM. The distinguishing factor between both categories is whether the vector length, i.e. the number of bits in a vector register is fixed (multimedia extensions) or not (vector architectures).

High performance and energy efficiency in combination with the demands of neural network applications has brought vector architectures again into the spotlight. Moreover, the bloating of *Instruction Set Architectures* (ISAs) is becoming more serious these days as increasing the vector length always means to add new instructions to the ISA [13], for instance the 32-bit x86 ISA from Intel has grown from 80 to about 1400 instructions since 1978, mainly due to SIMD. Here, vector architectures play out their advantage, i.e. without rewriting or recompilation of the software the same code runs on an embedded processor or a high-performance processor.

Following the vector architecture principle, the prominent RISC-V ISA [15] offers the *RISC-V "V" Vector Extension* (RVV) [3] which is currently in the final step of ratification. While we will see many RISC-V cores supporting RVV (and hence implementing a vector architecture) in the near future, leveraging the vector potential in the software is a very challenging task. Multiple development iterations when implementing/optimizing algorithms for RVV including verification, measurements, and evaluation are necessary. Moreover, algorithms usually serve a practical purpose in a bigger context and are therefore integrated into larger software systems. This leads to five practical challenges: (i) run-time results of algorithms that are deeply integrated into large software systems are often difficult to access and therefore hard to verify, (ii) instrumentation for run-time measurements is hard to integrate and maintain in existing software systems, (iii) instrumentation must be specifically implemented for every software system, (iv) run-time instrumentation and verification is specific for every software system which makes it hard to get comparable results, and finally (v) the performance of specific implementations may vary between different hardware platforms.

Contribution: In this paper, we propose *RVVRadar*[1] to tackle these challenges. RVVRadar provides a **standardized framework** to support programmers during the **development, verification, measurement and evaluation during the vectorization process of an algorithm**.

RVVRadar follows a bottom-up design approach. When a new algorithm has to be implemented for some target software system, it is added to the framework including a baseline implementation. In an iterative manner, programmers can add new optimized implementations to RVVRadar. When the framework is executed on a target hardware platform it automatically verifies and measures the performance of all implementations and generates standardized statistics. Based on these statistics the implementations can be optimized or a completely new implementation can be added.

RVVRadar provides a level of abstraction that makes it simple to evaluate the implementations on multiple hardware platforms.

GLSVLSI '22, June 6–8, 2022, Irvine, CA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9322-5/22/06…$15.00
https://doi.org/10.1145/3526241.3530388

[1]https://github.com/ics-jku/RVVRadar

This is especially useful since implementations might show diverse behaviors on different hardware platforms.

We demonstrate the advantages of RVVRadar for vectorization leveraging RVV on several practical relevant algorithms. In particular, we vectorized all filters used in the ubiquitous libpng library which increased the performance of these filters by a factor of up to 5.46. We released RVVRadar as well as the benchmarks[2] as open source software, and contributed our vectorized libpng filters to the libpng upstream.

2 RELATED WORK

Optimizing software to make better use of parallel processing (e.g. via SIMD, vector architectures, HW/SW partitioning) is a widely researched topic. Many works try to approach this optimization problem with automatic methods. [14] reviews notable approaches which translate binaries to custom hardware, thus shifting the most intensive parts of the binary to a highly parallel co-processor. If the processor contains a SIMD or vector unit, compilers can directly generate vectorized code [6]. Even if no SIMD or vector unit is present the compiler can emit optimized code that for example reduces loops by packing multiple operations into single instructions. Today automatic vectorization, which detects suitable structures in a program and generates vectorized code for them, is included in many compilers [7, 11]. However, automatic vectorization is not guaranteed to result in a higher performance. Other automatic approaches are fine-tuned to very specific applications (e.g. graph processing [10]) and thus are limited when applied to general programs.

Compared to the automatic approaches RVVRadar is not a "one-size-fits-all" technique. Even though automatic tools show good results in specific domains, a framework is needed for general use cases or domains not yet explored by the automatic techniques. RVVRadar provides this framework and to the best of our knowledge no such framework has been presented so far. With RVVRadar programmers can use their application and platform knowledge to vectorize their algorithms in an iterative manner.

3 RVVRADAR

In this section, we present RVVRadar and the RVVRadar vectorization workflow.

3.1 Problem Formulation

Vectorizing an *algorithm* is not a straightforward task. In particular, the configurability of the RISC-V vector extension opens up a wide range of options for the programmer. Exploring these options requires thorough evaluation of various *implementations* to find the best performing candidate. This evaluation has to be carried out not only for each implementation but also for a wide variety of *workloads* that test the performance in different scenarios. Finally, each implementation has to be executed many times to filter out measurement inaccuracies, since the run-time of an implementation is susceptible to side effects (e.g. from the operating system), and the run-times of all implementations must be aggregated in a report suitable for further analysis by the programmers. Handling all these tasks alone adds significant additional overhead to the development process.

We propose RVVRadar as an integrated framework that handles all of these steps with little additional work for the user. This way

[2]https://github.com/mschlaegl/libpng_rvv-doc

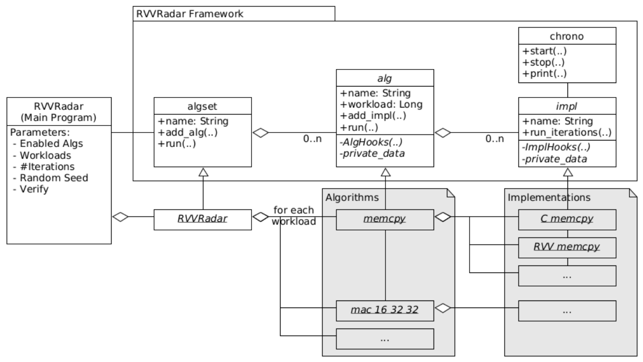

Figure 1: RVVRadar Structure Diagram

Table 1: RVVRadar Hooks

Hook	Description	Mandatory
algPreExec	Called before running the implementations (Create buffers, generate input/verification data)	
algPostExec	Called after running all implementations (Cleanup)	
implInit	Called before execution of first iteration	
implPreExec	Called before each iteration (e.g. cleanup for later verification)	
implExec	execution wrapper for specific implementation (this gets measured using chrono)	✓
implPostExec	Called after each iteration (e.g. verify result)	
implCleanup	Called after execution of all iterations	

RVVRadar supports the programmer in the four major phases of development, verification, measurement, and evaluation.

3.2 Design Criteria

Today, especially the ecosystem of embedded devices is growing tremendously. RVVRadar should be highly flexible to support as many devices and systems as possible. Therefore, we designed it as a C framework mainly targeted towards the GNU/Linux operating system. However, porting RVVRadar to other operating systems or even bare metal environments is possible with minor effort.

In principle, RVVRadar is not bound to a specific application domain. It therefore provides all the means to embed algorithms and feed them with the input data they require. RVVRadar makes heavy use of callbacks by providing hooks that can be used by programmers to inject custom code (e.g. for setting-up test data or verifying results) into the evaluation loop of RVVRadar.

In the following two sections we present the architecture and the evaluation loop of RVVRadar.

3.3 Architecture

In this section, we give an overview about the architecture including the major data structures of RVVRadar. The structure diagram of RVVRadar is shown in Figure 1. RVVRadar is used via the main program, also called RVVRadar in Figure 1 and shown on the left: It runs the evaluation and provides a *Command Line Interface* (CLI). The classes *algset*, *alg*, *impl*, and *chrono* build up the core of the framework. Using the *alg* class programmers can add new algorithms to RVVRadar (e.g. memcpy, multiply-accumulate). Its API provides functions to create instances and to create and add implementations (*impl*s) to it. Each algorithm has a name, a list of implementations, optional private data and the optional hooks as listed in the upper part of Table 1 to add specific functionality.

Please note the difference between an **algorithm** and its **implementations**. The algorithm is just the kind of problem that needs to be solved (e.g. memcpy: copy data from A to B) while there can be multiple implementations that actually solve this problem (e.g. a naive C memcpy, or an RVV-based implementation using a certain register grouping) which need to be evaluated and compared against each other.

The *impl* class features an API for creating these implementations. Like algorithms, implementations have names, private data and the hooks as shown in the lower part of Table 1. These hooks can be used to inject implementation-specific code before and/or after the execution of this implementation.

Additionally, each instance of *impl* contains an instance of *chrono*. The *chrono* class handles the run-time measurements and calculation of timing statistics. From multiple iterations of run-time measurements of implementations it calculates minimum, maximum, mean, variance, standard deviation, median, and a histogram.

By creating algorithms and assigning implementations to them, a tree is created which spans the design space RVVRadar should evaluate. The class *algset* groups multiple of these trees (one for each algorithm) together and serves as the top-level container for the evaluation loop.

3.4 Evaluation Loop

This section presents the evaluation loop of RVVRadar. The idea behind the evaluation loop is to provide an extensible solution that automatically runs and analyzes all algorithms/implementations. Besides the general operations of looping over all algorithms/implementations, executing, measuring, and reporting the results, the evaluation loop provides hook functions at critical points. The hooks allow integrating virtually every algorithm into RVVRadar since programmers can use them to adapt the evaluation loop to their needs.

A high-level description of the evaluation loop is shown in pseudocode 1. Before the core evaluation loop starts, the internal data structures are initialized in the construction phase (Line 1-4). This will fill the *algset* list with the algorithms. Each algorithm can be included in the list multiple times once for each workload. The algorithms contain all of their implementations that were created earlier using the APIs presented in Section 3.3.

Then, the main evaluation loops starts by iterating over all elements in *algset*. After an algorithm has been fetched from *algset*, the *preExec* hook of this algorithm is called (Line 6). This hook can be used to set up the environment shared by all implementations (e.g. buffers, parameters) and it also can be used to generate the input and verification data.

Once the environment is set up, the execution of all implementations starts. First, the *initHook* of the current implementation is called which can be used to set up implementation specific data (Line 8). Then, the implementation is evaluated multiple times (Line 9-14). The execution of one iteration of the implementation is handled by the *preExec* (Line 10), *exec* (Line 12), and *postExec* (Line 14) hooks that set up the data, execute the implementation, and perform verification.

After all iterations of one implementation have been executed, the measured results are used to calculate the statistics for this implementation (Line 15). These statistics are then written to files both as human-readable and machine-readable reports for later analysis. Also, at this point, implementation specific data can be cleared using the *cleanup* hook.

Finally, after all implementations have been evaluated, the *postExec* hook of the current algorithm is called which is used to clean up algorithm specific data.

Pseudocode 1 RVVRadar Evaluation Loop

```
 1: algset ← ∅
 2: for workload in workloads do                      ▷ construction
 3:     for alg in algs do
 4:         algset.append(create(alg, workload))
 5: for alg in algset do                              ▷ handle algorithms
 6:     call(alg.preExecHook)
 7:     for impl in alg do                            ▷ handle implementations
 8:         call(impl.initHook)
 9:         for all iterations do                     ▷ repeat measurement
10:             call(impl.preExecHook)
11:             startMeassurement()
12:             call(impl.execHook)
13:             endMeassurement()
14:             call(impl.postExecHook)
15:         calculateStats()
16:         outputHumanReadable()
17:         outputMachineReadable()
18:         call(impl.cleanupHook)
19:     call(alg.postExecHook)
```

3.5 Integrating a New Algorithm

The necessary steps to integrate a new algorithm in RVVRadar are described in this section.

For this, the algorithms and implementations shipped with RVVRadar can be used as a starting template. One example is *memcpy*, which performs a copy of a number of bytes from one memory location to another. It includes a very simple platform independent C implementation, an assembler RISC-V implementation using four integer registers and multiple RVV assembler implementations.

Initially, the *memcpy* directory has to be copied to a new directory with a name that corresponds to the new algorithm. The files *alg.h*/*alg.c*, which contain the code that assembles the algorithm and implementation data structures using the API from Section 3.3, must be updated to the new algorithm. The hooks used by the algorithm or the implementations should be also defined in these files.

In *impl_c.c.in* the baseline C implementation is defined. This file should be updated to hold the baseline C implementation of the new algorithm. RVVRadar treats the *impl_c.c.in* file differently since it compiles this file twice, once without and once with the auto-vectorization feature of the *GNU Compiler Collection* (GCC). All other implementations are stored in files matching the *impl_*.c* name format. New implementations can be added by creating new files starting with *impl_*.

Now the build system of RVVRadar can automatically build and link the new algorithm and its implementations to RVVRadar, but the algorithm is not yet selectable in the main program. To achieve this, the new algorithm has to be included in the *RVVRadar.c* program analogous to other existing algorithms.

4 CASE STUDY

In this section, we present a case study that shows how RVVRadar is used for developing optimized implementations leveraging RVV. We demonstrate the workflow of RVVRadar and in addition bring value to the open RISC-V ecosystem: We add RVV-optimized implementations to the open reference implementation of *Portable Network Graphic* (PNG), i.e. the library **libpng** [1]. Before we present the case study (Section 4.3) and further results (Section 4.4), we first give some general background on libpng (Section 4.1) followed by the experimental setup (Section 4.2).

4.1 Background on libpng

Since PNG is among the most common image formats on the web, libpng is very widely used, especially in Linux-based systems and therefore found in personal and enterprise computing as well as in mobile and embedded systems. The PNG format employs a lossless compression process in two stages [2]. The first stage is the pre-compression (or filtering stage). The basic idea of this is to exploit the fact that neighboring pixels have similar values and therefore storing the differences requires less space. The second stage is the compression (or deflation stage). It provides lossless compression using LZ77 and Huffman coding. In our case study we focus on the filtering stage and decompression.

PNG allows the use of different *filter methods* for the entire image, but only *filter method 0* is defined in the current PNG specification. *Filter method 0* allows selecting five different *filter types* for each row of the image: *None*, *Sub*, *Up*, *Average* and *Paeth*. Which filter is used for which row is selected by the compression application, usually using a heuristic. The decompression application executes the inverse of the *filter type* selected for each row.

libpng already provides infrastructure to support architecture specific implementations of *filter types*. This includes implementations for *Intel SSE2*, *ARM Neon*, *MIPS MSA*, and *PowerPC VSX* which can be enabled at build time or in some cases automatically at run-time. However, no implementation using RVV is available.

4.2 Experimental Setup

Since RVV is not ratified yet, currently RISC-V hardware capable of running Linux and supporting RVV is hardly available. However, one such board is the very recently released *Nezha* from RVBoards with an *Allwinner D1* SoC containing a *Xuantie C906 R1S0* RISC-V core. We use this board in our case study. The RISC-V core of the *Nezha* board supports RV64GCV and implements the *RVV Draft 0.7.1*. The vector instructions we needed for vectorizing the filter types can be directly transformed into their RVV 1.0 counterparts. For this, we have already integrated necessary mappings of RVV instruction mnemonics into RVVRadar.

To use RVVRadar it is necessary to specify a range of workloads to be processed by the implementations being measured. A workload corresponds to the number of elements processed by the implementations which in case of the *filter methods* is the number of pixels. Since all *filter type* algorithms have linear run-time complexity wrt. pixels times the constant number color channels, measurements on a single workload are sufficient for performance evaluation.

Each run-time measurement has an overhead caused by the instrumentation for the measurement itself. To minimize the impact of this overhead, the workload must be chosen such that the amount of time spend on running an implementation is much larger than the amount of time spent on the measurement. To fulfill this requirement we selected a single workload of 50 million elements (50 megapixels).

In the case study we focus on the average throughput of the developed implementations which is calculated by dividing the workload by the average run-time. RVVRadar provides two averaging methods for its run-time measurements, namely the arithmetic mean and the median. Since GNU/Linux is not a hard real-time system and therefore not deterministic we have to deal with run-time spikes. For this reason we use the median as it is more robust against such outliers.

4.3 PNG Paeth

Paeth is the most complex *filter type* used in PNG. It uses the difference of the *Paeth Predictor* [12] applied on the left, upper and left upper neighbor and the current pixel. The inverse is therefore to add the output of the *Paeth Predictor* to the current pixel.

The unoptimized C implementation of *Paeth* loops over all color channels per pixel per row. The goal is to parallelize this as much as possible using RVV.

Since the result of a pixel depends on the result of the previous pixel in a row, the maximum possible parallelization is limited to the number of color channels per pixel. This means that for RGB only three and for RGBA only four elements can be processed in parallel. The theoretical maximum performance gain is therefore limited to a factor of three or four, respectively.

To start the optimization process, the first step is to integrate the algorithm into RVVRadar as presented in Section 3.5. Thus, we add the unoptimized C implementation of *Paeth* mentioned above as baseline for verification and later comparisons to other implementations. Verification of further implementations is done by comparing the generated output against the baseline output given the same input.

Now, RVVRadar compiles the C implementation twice, once with and once without GCC‘s auto-vectorization. The measurement of the former forms the baseline for all further comparisons. The measurement of the latter can provide insight into the performance improvement that developers can achieve by using the out-of-the box vectorization from their compiler vendors.

In case of *Paeth* both measurements produce a throughput of 8.6 RGBA megapixels per second (MP/s). This shows that it is not possible for GCCs auto-vectorizer to optimize this code, and we take the 8.6 as our baseline for all further comparisons.

As next step, we developed a first RVV implementation. Basically, the C implementation is translated nearly directly to RVV. The only obvious optimization we introduce is to prevent multiple loads of the same data. This is achieved by reusing data (e.g. the pixel result) from the previous for the following iteration whenever possible.

The measurement of this implementation shows a throughput of 12.9 RGBA MP/s which is faster by a factor of about 1.5 than the baseline implementation.

As mentioned above, the maximum parallelization is limited by the number of color channels per pixel. It can be clearly seen, that the achieved performance gain is far from the theoretical maximum of factor 4 for RGBA. The reason for this is probably that the number of memory accesses dominates the run-time. Overall, the baseline implementation and the RVV implementation need to load and store the same amount of data. The potential advantage of RVV capable of loading/storing whole vectors at once can not be exploited for vectors with only four byte elements.

To address this, our next step is to test if the throughput improves when we increase the number of pixels loaded at once. The previous implementation iterated over all pixels in a row and loaded one pixel (four color channels) in each iteration. In this variant, we use a nested loop for processing. First a maximum possible number of pixels is loaded in a vector register (bulk load). The algorithm then iterates over all loaded pixels, calculates the results, and stores them similar as before. After that, another bulk load is performed. This is repeated until the whole row is processed.

Using RVV this is realized in the following way: For the load, the vector length is set to the full vector register length. This means, that on the used board 16 color channels, corresponding to four RGBA pixels are loaded in the vector register. When iterating over the color channels of a pixel, the vector length is reduced to the number of color

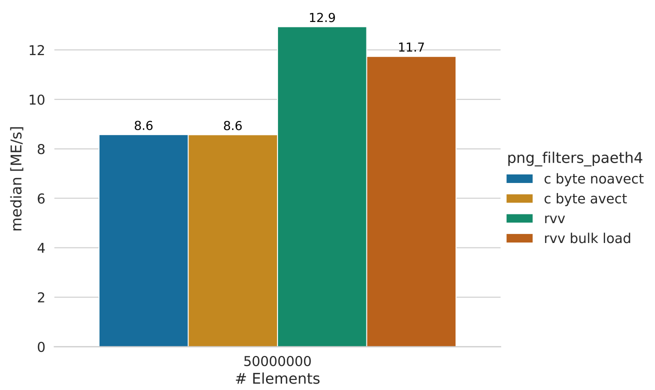

Figure 2: Median Throughputs of different Implementations of inverse PNG Paeth for RGBA Pixels on Allwinner D1

channels, so that only the first four elements of the vector register are processed. After each processed pixel, the next pixel (next four values) is shifted down by using `vslidedown.vx`. If no shift is possible, the next four RGBA pixels are loaded. As written above, this is repeated until the whole row is processed.

Running this implementation brought a throughput of 11.7 RGBA MP/s. The throughput is greater than the baseline by a factor 1.36. However, it is lower than the first RVV implementation which improved the performance by a factor of 1.5 compared to the baseline. It seems that the additional overhead (nested loop, shift) outweighs the benefit of the bulk load.

Figure 2 provides a summary, i.e. the median throughput of all implementations for *png_filter_paeth* on RGBA pixels. The X-axis shows the number of elements corresponding to RGBA pixels processed in one iteration and the Y-axis shows the median throughput in million elements per second which corresponds to RGBA MP/s. The different implementations are color-coded and listed in order of development in the legend on the left. *c byte noavect* and *c byte avect* are the C implementations without and with using GCCs auto-vectorization, respectively. *rvv* is the initial RVV implementation which loads and processes one pixel per iteration. *rvv_bulk_load* is the second RVV implementation which loads four pixels at once.

Based on the results of this case study we select the initial RVV implementation *rvv* (green bar in Figure 2) to be integrated in libpng.

Overall, the case study clearly demonstrates that the programmer is strongly supported by RVVRadar and hence can focus on the actual goal of optimizing algorithms.

4.4 More Algorithms

In the same way as in the case study just described, we used RVVRadar to optimize implementations for several more algorithms. Table 2 summarizes the results, i.e. the respective maximum speedup achieved in comparison to the unvectorized baseline C implementation is given. The *memcpy* algorithm is a "simple" copy of elements from one memory location to another. The two algorithms starting with *mac* are different variants of multiply-accumulate operations. The first multiplies two fields with 16bit values and adds them in-place to a given 32bit value field, while the second multiplies two fields with 8bit values, then adds a field with 16bit values and saves the result in a dedicated 32bit result field. The algorithms starting with *png_filters* are the PNG *filter types* for three (RGB) or four (RGBA) color channels. Again, we used a workload of 50 million elements and the median over multiple run-time measurements.

Table 2: Achieved speedups using RVVRadar

Algorithm	Speedup
memcpy	6.31
mac32bit += 16bit * 16bit	2.04
mac32bit = 16bit + 8bit * 8bit	3.99
png_filters_up3	5.43
png_filters_up4	5.43
png_filters_sub3	2.19
png_filters_sub4	3.07
png_filters_avg3	1.55
png_filters_avg4	2.07
png_filters_paeth3	1.13
png_filters_paeth4	1.51

Although the results were obtained on a low-end embedded RISC-V SoC, using RVVRadar we were able to leverage the potential of RVV. For libpng we achieved speedups of up to 5.43 (2.80 on average); overall we obtained 6.31 (3.16 on average). Moreover, our achieved speedups on the individual algorithms correspond to their expected parallelization potential.

5 CONCLUSION

In this paper, we presented the framework RVVRadar, which supports programmers in development, verification, measurement and evaluation during the vectorization process of an algorithm. We have demonstrated that the programmer can focus on the actual vectorization goal and significantly benefits from the infrastructure of RVVRadar to eventually find the appropriate implementation.

In terms of concrete results, we vectorized all filter methods of the widely-used libpng leveraging the RISC-V vector extension RVV. Our final implementations of the filter methods devised using RVVRadar achieve speedups of up to 5.43. We made the RVV-based libpng and RVVRadar available as open-source on GitHub.

For future work, we plan to integrate advanced verification techniques like EPEX [8] in RVVRadar.

ACKNOWLEDGMENTS

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

REFERENCES

[1] 2022. libpng. http://www.libpng.org/pub/png/libpng.html.
[2] 2022. Portable Network Graphics (PNG) Specification (Second Edition). https://www.w3.org/TR/PNG.
[3] 2022. RISC-V V vector extension. https://github.com/riscv/riscv-v-spec.
[4] Roger Espasa, Mateo Valero, and James E. Smith. 1998. Vector Architectures: Past, Present and Future. In *ICS*. 425–432.
[5] M.J. Flynn. 1966. Very high-speed computing systems. *IEEE* 54, 12 (1966), 1901–1909.
[6] F. Franchetti, S. Kral, J. Lorenz, and C.W. Ueberhuber. 2005. Efficient Utilization of SIMD Extensions. *Proc. IEEE* 93, 2 (2005).
[7] Inc. Free Software Foundation. 2021. Auto-vectorization in GCC. https://gcc.gnu.org/projects/tree-ssa/vectorization.html.
[8] Lucas Klemmer and Daniel Große. 2021. EPEX: Processor Verification by Equivalent Program Execution. In *GLSVLSI*. 33–38.
[9] R.B. Lee. 1997. Multimedia extensions for general-purpose processors. In *SiPS*. 9–23.
[10] Jie Lin, Qingbo Wu, Yusong Tan, Jie Yu, Qi Zhang, Xiaoling Li, and Lei Luo. 2017. MicRun: A framework for scale-free graph algorithms on SIMD architecture of the Xeon Phi. In *ASAP*.
[11] LLVM. 2021. Auto-vectorization in GCC. https://llvm.org/docs/Vectorizers.html.
[12] Alan W. Paeth. 1991. II.9 - IMAGE FILE COMPRESSION MADE EASY. In *Graphics Gems II*, James Arvo (Ed.). Morgan Kaufmann, 93–100.
[13] D. Patterson and A. Waterman. 2017. SIMD instructions considered harmful. https://www.sigarch.org/simd-instructions-considered-harmful.
[14] Nuno Paulino, João Canas Ferreira, and João M. P. Cardoso. 2020. Improving Performance and Energy Consumption in Embedded Systems via Binary Acceleration: A Survey. *ACM Comput. Surv.* 53, 1, Article 6 (feb 2020), 36 pages.
[15] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley.