Identifying Hardware Optimizations for Neural Network Inference using Virtual Prototypes

Jan Zielasko
Institute of Computer Science
University of Bremen
Cyber-Physical Systems
DFKI GmbH
Bremen, Germany
Jan.Zielasko@DFKI.de

Rolf Drechsler
Institute of Computer Science
University of Bremen
Cyber-Physical Systems
DFKI GmbH
Bremen, Germany
drechsler@uni-bremen.de

Abstract—**Identifying the optimal hardware configuration for neural network inference on ultra-low-power edge devices is critical for reducing cost and maximizing the performance of smart applications. Tailoring hardware designs to specific applications significantly improves resource utilization. However, locating profitable optimization points in complex workloads remains challenging, particularly when critical code lies outside the main compute kernels.**

We present an approach based on a RISC-V virtual prototype to systematically identify fine-grained hardware optimization opportunities. The virtual prototype models the entire hardware platform, including accelerators, while remaining fast and accessible. Combined with a custom execution-trace compression and analysis framework, it enables the capture and processing of billions of executed instructions.

Applied to representative edge artificial intelligence workloads from the MLPerf Tiny benchmark suite, our approach successfully identifies promising optimization opportunities beyond the matrix multiplication kernel that are non-trivial to detect from either the source code or gate-level analysis. We further validate our method using typical embedded workloads from the Embench IOT 2.0 suite, demonstrating its applicability to a wide range of embedded workloads.

Index Terms—**Virtual Prototyping, Neural Networks, Hardware Optimization, ASIC, RISC-V, Tracing.**

I. Introduction

Recent advances in AI have increased the demand for running complex applications such as *Neural Networks* (NNs) directly on resource-constrained devices e.g., for sensor data processing on embedded systems or *Internet of Things* (IoT) nodes [1]–[3]. While *General-Purpose Processors* (GPPs) are commonly used as the computational core for *System-on-Chip* (SoC) designs, they are inefficient for NN inference. To meet the strict energy and performance requirements, GPPs must be tailored to specific applications. However, identifying profitable optimization points in complex workloads remains a challenge.

Existing approaches address this challenge at different abstraction levels. For example, [4] proposes a compiler workflow to optimize applications for the target hardware. At the architecture level, automated mapping flows like [5] generate suitable hardware accelerators, while [6] uses in-memory compute to accelerate matrix multiplications. While effective, these methods are inherently limited by their abstraction boundaries and often dependent on optimizations from other levels.

Virtual Prototypes (VPs) combine these perspectives by providing an executable model of the entire *Hardware* (HW) platform including peripherals and accelerators [7]. As such, VPs enable early *Software* (SW) development and design space exploration before any real HW is built, and achieve simulation speeds orders of magnitude faster than RTL [8]. The RISC-V Opt VP *(Opt-VP)* [9] extends this concept to application-driven hardware optimization. It traces the execution at the instruction level and analyzes every encountered instruction sequence, identifying a wide range of optimization candidates for typical edge applications from the Embench$^{\text{TM}}$ 1.0 suite[1].

In this paper, we extend this VP-driven approach to the domain of neural network inference. Unlike traditional embedded workloads, inference is dominated by highly regular kernels such as matrix multiplication, making it harder to expose other optimization opportunities. We analyze representative edge AI applications implemented with *TensorFlow Lite for Microcontrollers*[2] and the *MLPerf Tiny* [3] benchmark suite. Our results show that the approach uncovers both expected candidates (multiply-accumulate) and previously overlooked instruction patterns outside the main kernels, which are non-trivial to find from either the source or gate-level. The extended tool[3] and all *Machine Learning* (ML) benchmarks [3] are available as open-source.

Contributions:

- We apply VP-driven instruction-level analysis to realistic NN inference and embedded workloads.
- We implement the tracing and analysis extensions to handle complex analysis tasks.
- We demonstrate that the approach identifies optimization candidates beyond matrix multiplication.

This work was supported by the German Federal Ministry of Research, Technology and Space (BMFTR) within projects ECXLplus (grant no. 01IW24001) and FAIRe (grant no. 01IS23074).

[1]https://www.embench.org/
[2]https://github.com/tensorflow/tflite-micro
[3]https://github.com/agra-uni-bremen/opt-vp

II. Related Work

Optimizing the performance of applications on embedded devices lies at the intersection of static compiler optimization, runtime analysis, *Instruction Set Architecture* (ISA) design and HW/*Application-specific Integrated Circuit* (ASIC) design. Each domain tries to find optimizations on different levels of abstraction with different goals (code size, energy, latency, area) and the results at one level often depend on or limit what can be achieved at others. Here we discuss how existing work addresses the problem at different abstraction levels.

A. RISC-V

A key component to consider when designing a new system is the ISA as it defines the set of instructions that the hardware supports. While designing a fully custom ISA is possible, it is time-consuming and requires a lot of expertise. Instead, extending an existing ISA is a more practical approach. However, not all ISAs are equally suited for this task. RISC-V [10] is an open-standard ISA that has gained traction in industry and academia in recent years. It provides a foundation for modular processor design and customization with its optional standard extensions and support for custom instructions, with a minimal implementation only requiring 45 instructions [11]. For these reasons many of the tools mentioned below target RISC-V.

B. Identifying Hardware Optimizations

Identifying profitable hardware optimizations is difficult and existing approaches often target a specific type of optimization. One of the most substantial performance improvements for any application can be achieved by choosing the right (application and use case specific) compiler flags [12], as this can drastically reduce code size, energy consumption, and execution time. However, compiler optimizations are inherently limited by the ISA of the target platform. Enabling the RISC-V Compressed extension, for example, can reduce code size by up to 25 % [13]. LLVM [14] tries to abstract away some of these limitations by providing a common intermediate representation that is independent of the hardware.

Some approaches try to directly map high-level operations to hardware primitives. FP-DNN [5], for example, automatically maps high-level deep neural network models directly to hardware implementations synthesizable on an FPGA.

A notable example that uses cross-domain information is SIZALIZER [15]. The tool uses a multi-layer analysis framework, combining the output of multiple existing tools, for the co-design of embedded C/C++ applications and RISC-V instruction set extensions. Specifically it uses the LLVM intermediate representation, the binary executable, and runtime information to increase code density and reduce the memory footprint. Compared to the Opt-VP, SIZALIZER focuses on static and dynamic code size reduction, while the VP approach targets a wider range of optimizations. Incorporating the LLVM Intermediate Representation into the VP analysis framework is an interesting direction for future work.

Recently a similar tool called ARISE [16] was proposed. ARISE automates the generation of RISC-V instruction set extensions by analyzing assembly patterns based on metrics that target code size and instruction count. It uses the ISA description language CoreDSL to generate the extension specifications. For applications in the Embench set it can reduce the number of executed instructions by 7.39 % on average.

C. Application Specific Hardware Design

While some of these approaches automatically generate hardware designs, most separate the analysis and design steps. Here we discuss two approaches that focus on enabling ASIC design and design space exploration.

Codasip Studio is a processor design toolset that uses the CodAL language to describe custom instructions. From this CodAL description, it automatically generates an LLVM C/C++ compiler backend that can utilize the custom instructions, as well as other outputs like executable models and RTL. It provides a convenient way for design space exploration of custom instructions and instruction set extensions. The results from the Opt-VP analysis can be used as input for Codasip Studio to evaluate the identified optimizations as it also targets RISC-V processors. We tested this workflow using a variant of a multiply-accumulate instruction identified by the VP.

A popular industry example for designing application-specific hardware is the Tensilica Xtensa [17] processor family from Cadence. By using the high-level Tensilica Instruction Extension language, designers can define custom instructions and hardware accelerators. It also applies compiler and simulator driven analysis to identify bottlenecks in the application to guide the design of optimizations. However, the tool is proprietary and not publicly available while targeting their own ISA. Codasip Studio on the other hand is available for academic use and targets the open RISC-V ISA.

In contrast to these works, our approach is fully open source and leverages a virtual prototype to derive optimization candidates dynamically from the full system execution. Additionally, the custom trace representation allows for a faster and more complex analysis compared to simple hotspot detection.

D. Virtual Prototypes

In the classical design flow of digital systems, a significant amount of time is spent on the development of physical prototypes, while verification and testing is only possible after it is finished. To accelerate the time-to-market, *Instruction Set Simulators* (ISSs) can be used to facilitate functional verification and early SW development. However, ISSs are primarily designed for speed and often lack accuracy and flexibility for system-level tasks like design space exploration or performance analysis [18]. SystemC based virtual prototypes like the RISC-V VP [7], address these limitations by providing an executable model of the entire hardware platform, that is able to simulate intricate HW/SW interactions including peripherals and accelerators. They achieve simulation speeds orders of magnitude faster than RTL, while still being accurate enough for system-level tasks.

The Opt-VP [9] extends this concept to application-driven hardware optimization by extending the RISC-V VP with instruction-level tracing and analysis capabilities.

III. Methodology

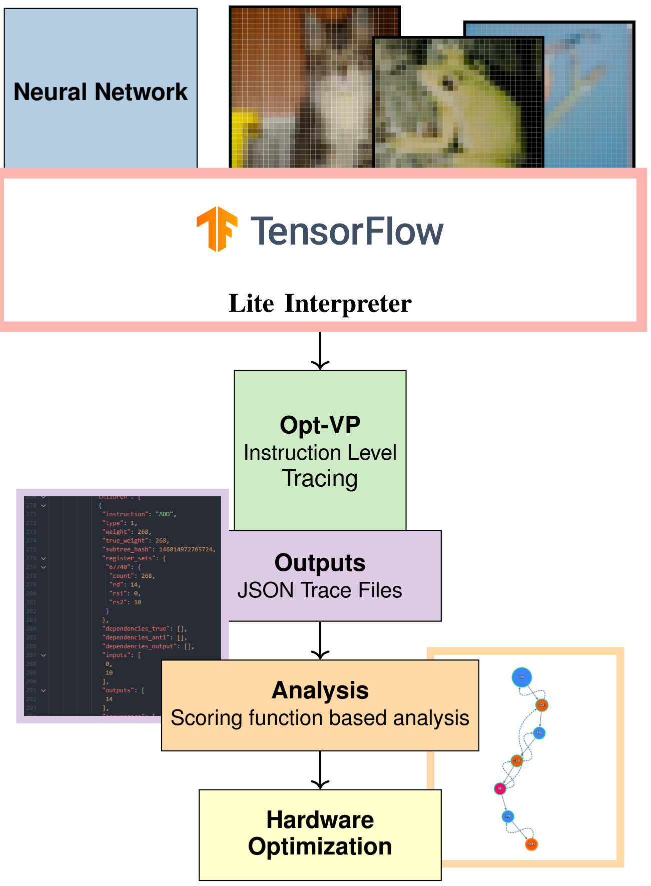
In this section we describe the VP-driven approach to identify hardware optimizations and our proposed intermediate trace representation. This work is based on the Opt-VP proposed in [9] and extends the framework with an improved tracing and analysis infrastructure to handle more complex workloads and analysis tasks.

A. Framework Overview

The VP models a configurable RISC-V RV32IMAC core and a set of peripherals. Figure 1 shows an overview of the workflow using the tool to analyze an image classification application. In the first step, it is executed on the VP, which traces every executed instruction, its parameters, effects on the system and SystemC transactions, and stores it in a compact bounded execution tree representation in memory. It creates a separate tree for each instruction, capturing all sequences starting with that instruction up to a fixed depth k. This representation captures local execution contexts without requiring storage of the full instruction trace, which would be infeasible for multi-billion-instruction workloads. Limiting the scope to individual sequences also allows for more sophisticated analysis techniques during the execution, such as taint tracking or data dependency analysis. During our experiments with different applications, we discovered that a bound of 50 is sufficient to discover all relevant sequences for almost any application. Further analysis shows, that in practice, a bound of 10 is enough to discover all relevant sequences for smaller applications (e. g., the Embench set). If the bound is set too low, the tool suggests increasing it to uncover additional sequences.

After the execution is finished, the execution trees are analyzed using a configurable scoring function that considers metrics such as sequence length, execution frequency, dependencies, inputs or memory accesses. The goal is to identify instruction sequences that represent promising hardware optimization candidates, e.g., fused instructions, custom instructions, or accelerator kernels. The analysis yields a ranked set of recurring instruction sequences, based on the chosen scoring function. Unlike static or compiler-only approaches, Opt-VP derives these candidates directly from dynamic execution of the full system, including peripheral accesses and library code, ensuring discovered candidates are relevant to the actual workload and to avoid optimizing for edge cases. To further improve coverage and therefore the profitability of individual optimizations, similar sequences can be merged using the tool proposed in [19].

The tracing module incurs considerable runtime overhead compared to the baseline VP, which depends on the tracing configuration and tree depth (i.e., the maximum length of instruction sequences considered for optimization). For a depth of 20 with full memory and dependency tracking the VP runs around 100 times slower compared to the baseline VP. ML applications like image classification often incur additional overhead due to their high number of different memory accesses. By utilizing appropriate C++ data structures and optimizing the algorithms for common access patterns, the



```
      children": [
        {
         "instruction": "ADD",
         "type": 1,
         "weight": 268,
         "true_weight": 268,
         "subtree_hash": 146814972765724,
         "register_sets": {
          "67740": {
           "count": 268,
           "rd": 14,
           "rs1": 0,
           "rs2": 10
          }
         },
         "dependencies_true": [],
         "dependencies_anti": [],
         "dependencies_output": [],
         "inputs": [
          0,
          10
         ],
         "outputs": [
          14
         ],
```

Fig. 1. Analysis Workflow for analyzing a ML application using the Opt-VP

overhead was reduced significantly compared to the original implementation, resulting in tracing speeds of greater than 200 k instructions per second for $k = 20$. Memory usage remains reasonably low, even for greater depths and large applications thanks to the compression into execution trees. For a $k = 20$, the tool uses less than 6 GB of memory to trace any of the MLPerf applications. During execution the usage stays relatively constant and is largely independent of the number of executed instructions, as repeated executions of the same code map to identical paths in the trees.

The tree representation also allows for an efficient analysis of all instruction sequences and can brute-force evaluate all possible sequences for a k greater than 50 in less than 3 seconds even for large applications. This allows for rapid exploration of different scoring functions and analysis techniques.

B. JITR: JSON Intermediate Trace Representation

To enable a systematic analysis of the trace data independently of the VP, we developed *JITR*, a hierarchical intermediate trace representation and added an export function to the VP. To capture the hierarchical structure of the execution trees, the format is based on JSON objects. This format contains all accumulated execution tree data including dependencies. Figure 2 shows an overview of the format. Every node contains the *instruction* name, its *weight* (i.e., how often it was executed), the *registers* read and written by the instruction, the *dependencies* and all child nodes. The *true_weight* only counts unique paths,

```
{
  "instruction": (string) instruction name
  "weight": (uint)   number of occurrences
  "true_weight": (uint)  only unique paths
  "register_sets": I/O List {
      "PC": (uint)    program counter
      "count": (uint) number of occurrences
      "rd":  (int) -1 if not used
      "rs1": (int) -1 if not used
      "rs2": (int) -1 if not used
  }
  "dependencies_true":   (uint[]) RAW
  "dependencies_anti":   (uint[]) WAR
  "dependencies_output": (uint[]) WAW
  "subtree_hash": (uint) hash of the sequence
  *"accesses":   ([pc, [[addr, region]]])[]
  *"occurrence": ([[int]]) lifetime occurrences
  *"direction": (int) branch direction
  "children": (node[]) child nodes
}
```

Fig. 2. JSON intermediate trace representation

which is relevant for sequences that are not prefix free (e.g., ADD→LW→ADD→LW). Optional and type dependent attributes are marked with an asterisk. E.g., accesses to memory is only relevant for load/store instructions. Each load/store instruction includes information on all accessed memory addresses and their corresponding regions (e.g., stack, heap, frame), which enable fine-grained identification of potential memory hazards such as overlapping or sequential memory accesses. This is essential for distinguishing parallelizable memory operations from those requiring strict serialization.

C. Scoring Functions

Scoring functions are a key component of the analysis framework as they determine the most profitable optimization candidates. They essentially rank every encountered instruction sequence based on a set of metrics. The available metrics consist of the attributes contained inside the JSON intermediate representation as well as the following additional parameters:

- **sequence** of instructions from root to this node
- **length** of the sequence
- **score_modifier**: passed from parent to child
- **score_mult**: passed from parent to child

The scoring function itself is an arbitrary C++/Python function defined by the user that can freely combine these metrics. Figure 3 shows a simple scoring function that identifies the sequences with the highest coverage and at least length 2 (line 2), that contain no branches (i.e., excludes any instruction mnemonic matching `B.*`) except for the last instruction. To evaluate the scoring function, it is applied depth-first to every node in the execution trees. The returned score is then used to rank the sequences. The special parameters *score_modifier* and *score_mult* can be used to propagate information from parent to child nodes, e.g., to penalize sequences containing certain instructions. The Opt-VP can evaluate scoring functions written in C++ and allows for dynamic reloading of scoring functions at runtime.

```python
def sc_fn(params, score_modifier, score_mult):
    score = params.true_weight*(params.length-1)
    score += score_modifier
    score *= score_mult
    if params.instruction.startswith('B'):
        score_mult *= 0.0
    return score, score_modifier, score_mult
```

Fig. 3. Example scoring function

D. Custom Instructions

Based on the identified sequences we can define hardware optimizations that replace or accelerate the original sequence during execution. To design a custom instruction, we evaluate the dependencies, memory accesses and instructions of the sequence. For example, a sequence consisting of only arithmetic instructions with no true dependencies can be implemented as a single custom instruction, that executes all operations in parallel. If the sequence contains multiple memory accesses, we need to ensure that they do not overlap (i.e., no dependencies). Critical instructions like branches or system calls can be handled at this point using techniques like trace scheduling [20], but a better approach is to exclude them through the scoring function. When trying to parallelize instructions, we overestimate the dependencies to ensure correctness and increase coverage. It would be possible to apply a more aggressive approach with a tradeoff between parallelism and coverage.

To estimate the performance impact of optimizations, we developed a Just-In-Time accelerator interface for the VP that can dynamically replace identified instruction sequences with a single custom instruction. However, it is possible to calculate the performance impact of an optimization statically from the execution trees without re-executing the application. For this reason we introduced the *true_weight* attribute that allows us to calculate the exact coverage of a sequence. Based on the coverage and the saved cycles per replacement, we can estimate the total performance impact of an optimization. This enables rapid exploration of different optimizations.

IV. Experimental Setup

To evaluate our approach, we applied the extended Opt-VP to a diverse set of embedded and ML workloads. The resulting trace data was analyzed using the built-in analysis framework and external Python scripts that operate on *JITR*.

We selected benchmarks from three sources:

- **EmbenchTM IOT 2.0** [21]: a standard suite for evaluating embedded processors, representing typical edge workloads such as cryptography, signal processing, and compression and since version 2.0 also convolutional NN.
- **MLPerf Tiny** [3]: a widely adopted benchmark suite for edge AI inference, covering anomaly detection, keyword spotting, visual wake words, and image classification.
- **TensorFlow Lite for Microcontrollers** [22]: edge AI applications, e.g., *sine wave regression* and *person detection*.

All applications were compiled using the standard settings from their respective repositories. While the current TF Lite RT framework does not officially support a RISC-V board on its

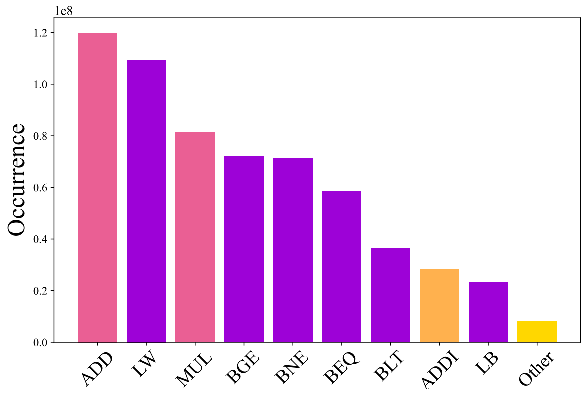


Fig. 4. Instruction distribution for image classification

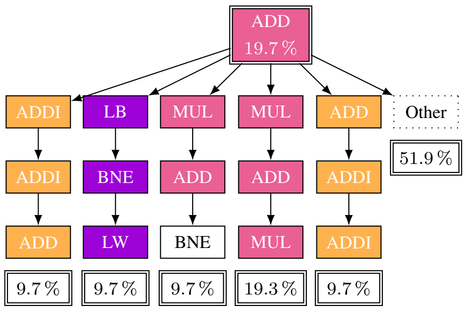


Fig. 5. Execution tree with different application-specific patterns: ☐ Other, 🟥 MAC, 🟧 Addition, 🟪 Branching

website, it is possible to build the framework for RV32IMA with slight modifications. MLPerf Tiny on the other hand required a complete rewrite of the build system and interpreter integration. For consistency, we integrated the MLPerf Tiny applications into the same build system as TF Lite RT.

Our target for the analysis is the identification of on chip custom instructions. For this reason we configured the VP to represent a minimal RISC-V RV32IMAC core with 16 MB of RAM and enabled full tracing with a tree depth of 20.

V. Results

In this section we present the results of our analysis for the selected benchmarks. The first section focuses on the analysis of a single image classification application from MLPerf Tiny to demonstrate the capabilities of the tool. The following section presents the combined results of all selected benchmarks.

A. *Case Study: Image Classification*

We demonstrate the tool in a case study using the image classification reference model from *MLPerf Tiny*. It is a ResNet8 model trained on the CIFAR10 dataset able to classify low resolution images of 10 different categories. Running a single inference cycle executes approximately 6.08×10^8 instructions, simulating 23.2 seconds. The execution time of the simulation takes around 4 hours for a depth of 20. In comparison, the md5sum benchmark takes around 2 minutes to execute and trace. Analysis of the trees, i.e., exhaustively exploring all sequences takes less than 1 second. Figure 4 shows the distribution of instructions among all sequences. `ADD` (19.7 %) and `LW` (17.9 %) were most frequently executed with only 9 instructions amounting to 98.7 % of total execution. Figure 5 represents an excerpt from the tree containing all sequences starting with an `ADD` instruction. As expected, by analyzing the trees we are able to identify multiple different variations of MAC operations. Two of which are contained in the `ADD` tree highlighted in 🟥. 🟧 highlights a chain of `ADD` operations related to the matrix multiplication. The instructions highlighted in 🟪 contain part of the most interesting sequence discovered by our analysis. They are part of a set of longer sequences containing a number of branch and load instructions depending on a single value from memory. This makes the sequence a good candidate for parallel execution as a new custom instruction.

Next, we design custom instructions based on the sequences. For the sequence `ADD`$\rightarrow$`ADD`$\rightarrow$`ADDI`$\rightarrow$`ADDI`$\rightarrow$`ADD`$\rightarrow$`MUL`$\rightarrow$`ADD`, we managed to design a custom instruction with 3 stages saving 4 cycles per execution (compared to the original 9 cycles):

- Stage 1: `ADDI, ADDI, ADD, ADD` (4 parallel, 1 cycle)
- Stage 2: `ADD, MUL` (2 parallel, 3 cycles)
- Stage 3: `ADD` (1 cycle)

The sequence has a coverage of 13.3 % and a speedup of 1.8. Using Amdahl's law to calculate the overall performance impact results in a speedup of 6 %.

For the other sequence `ADD` $\rightarrow$ `LB` $\rightarrow$ `BNE`$\rightarrow$ `LW` and its extension with length 16, we can design a conditional move instruction that loads a value from memory. The theoretical speedup for the full sequence is 31.1 %, but the large number of branches make this unfeasible in actual hardware. A two stage implementation of the first 4 instructions results in a speedup of 2.0 %.

To evaluate the impact of different metrics on the resulting candidates and their expected performance impact, we analyzed the trace with a range of different scoring functions, each maximizing coverage while applying stacking constraints:

- SF1: minimum length of 2
- SF2: + no branch instruction inside the sequence
- SF3: + discourage consecutive ADD/MUL instructions

For each scoring function we take the top 200 candidates with a Jaccard similarity threshold of 30 %. Figure 6 shows the differences in sequence length between the scoring functions. SF2 and SF3 result in sequences of similar length, with SF3 discovering slightly longer sequences. SF1 on the other hand identifies a larger number of long sequences as a result of the high number of loops during the execution. Comparing the coverage of instructions in the identified sequences shows that the inclusion of branch instructions leads to an increase of 50 % for the top candidates (around 38 % total coverage) and around 3000 % for the average case. This result indicates that accelerating sequences that include branches can significantly improve performance, however these types of sequences are more difficult to implement as custom instructions.

B. *Multi Domain Results*

To compare the effectiveness of our approach across different domains, and analyze the optimization potential of typical embedded workloads, we applied our analysis to a range of benchmarks from Embench 2.0, MLPerf Tiny and TensorFlow Lite for Microcontrollers. All applications were analyzed using the same scoring function (Figure 3) and we selected the top 50 candidates with a Jaccard similarity threshold of 30 %. Figure 7

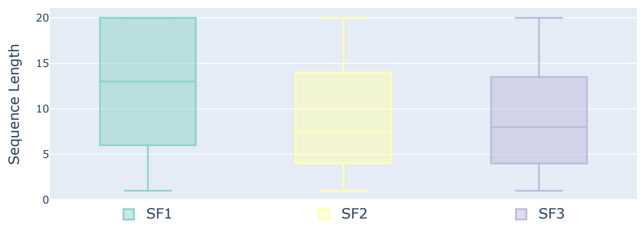


Fig. 6. Length distribution based on different scoring functions

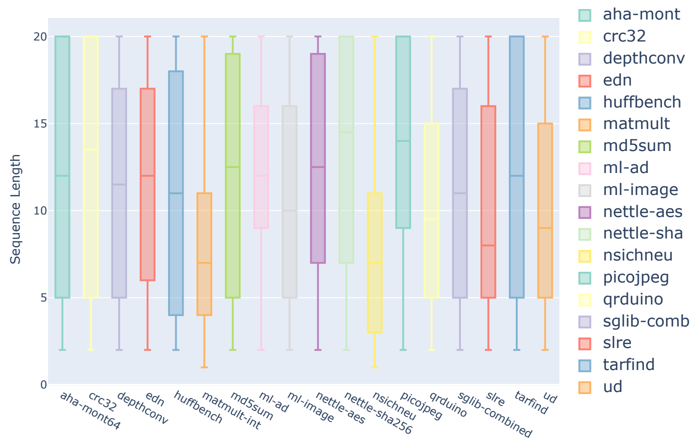


Fig. 7. Length distribution of identified sequences across different domains

shows the length distribution of the identified sequences for a selection of benchmarks. While the image classification application (*ml-image*) entry shows numerous long sequences, the length is uniformly distributed starting at a length of 5. The results for anomaly detection (*ml-ad*), in contrast, show a distribution skewed towards the upper limit. Extending the analysis bound could yield better results, however $k = 20$ is already quite high for designing custom instructions. This result means, that an off chip accelerator designed to implement larger sequences is likely better suited for the application. On the other hand smaller custom instructions can already achieve good results for the image-classification application. Similar examples of this are picojpeg and tarfind, which show an even stronger skew towards longer sequences. The results for *depthconv*, *edn*, *nsichneu*, *sglib-combined* and *slre* show a distribution centered around a length of 12, indicating that medium size custom instructions and small accelerators should perform well. Figure 8 and Figure 9 show the coverage and expected speedup. *Ml-ad* and *matmult-int* show by far the highest peak coverage with over 80 % of the total execution covered by a single sequence for *ml-ad*. The same sequence also achieves an estimated speedup of 60 %. *Picojpeg* and *slre* on the other hand are barely profitable with very low coverage and speedup.

VI. Conclusion and Future Work

In this paper we presented an extension to the VP-driven approach, and analyzed a range of ML and embedded applications. We demonstrated that we are able to identify typical ML optimizations like MAC operations, design custom RISC-V instructions and estimate their performance impact. Additionally, we discovered that the tool can be used to identify

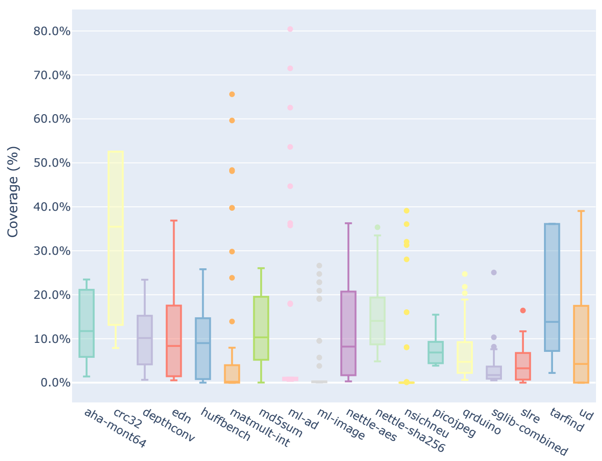


Fig. 8. Coverage of identified sequences across different domains

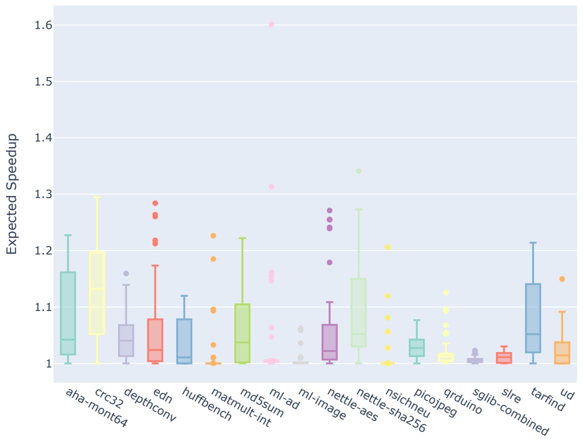


Fig. 9. Expected speedup of custom instructions ignoring hardware complexity except branches

additional HW optimizations that are non-trivial for other analysis approaches. Using the execution tree representation and a set of scoring functions, we are able to perform rapid design space exploration for custom instructions for a broad range of applications. Our evaluations show promising sequences with high coverage and expected speedup up to 60 % for single custom instructions. These results demonstrate that VP-driven analysis can guide selective ISA and microarchitectural extensions beyond conventional hotspot detection. To allow for analysis tasks independently of the VP, we proposed a hierarchical JSON trace representation and tree export.

For future work we plan to evaluate the accuracy of our performance estimation by measuring the impact of proposed custom instruction on the RTL level using e.g., SpinalHDL [23]. Furthermore, we plan to implement a standalone analysis frontend that can be used to process the generated trace files and make it easier to incorporate additional data sources. Especially the integration of the high-level LLVM Intermediate Representation could solve the register renaming problem and allow for more aggressive merging of sequences. Another interesting direction is the automated integration of custom instructions into the VP and compiler toolchain. This could be done using either SAIL [24], CoreDSL or CodAL.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, “A new golden age for computer architecture,” *Commun. ACM*, vol. 62, no. 2, p. 48–60, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3282307

[2] C. B. et al., “Terminology for Constrained-Node Networks,” RFC 7228, May 2014. [Online]. Available: https://www.rfc-editor.org/info/rfc7228

[3] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau *et al.*, “Mlperf tiny benchmark,” *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

[4] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, “Tvm: An automated end-to-end optimizing compiler for deep learning,” 2018. [Online]. Available: https://arxiv.org/abs/1802.04799

[5] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, “Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates,” in *FCCM*, 2017, pp. 152–159.

[6] W. Wan, R. Kubendran, C. Schaefer, S. B. Eryilmaz, W. Zhang, D. Wu, S. Deiss, P. Raina, H. Qian, B. Gao, S. Joshi, H. Wu, H.-S. P. Wong, and G. Cauwenberghs, “A compute-in-memory chip based on resistive random-access memory,” *Nature*, vol. 608, 2022.

[7] V. Herdt, D. Große, P. Pieper, and R. Drechsler, “RISC-V based virtual prototype,” *Journal of Systems Architecture*, vol. 109, 2020.

[8] F. Ghenassia, *Transaction-Level Modeling with SystemC*. New York: Springer, 2010.

[9] J. Zielasko and R. Drechsler, “Virtual prototype driven application specific hardware optimization,” in *FDL*. IEEE, 2023, pp. 1–8.

[10] *The RISC-V Instruction Set Manual*, RISC-V International, 2017, version 20250508, accessed: 2025-09-13. [Online]. Available: https://github.com/riscv/riscv-isa-manual/

[11] A. Waterman, Y. Lee, R. Avizienis, H. Cook, D. Patterson, and K. Asanovic, “The RISC-V instruction set,” in *IEEE Hot Chips 25 Symposium (HCS)*. IEEE, 2013.

[12] J. Pallister, S. Hollis, and J. Bennett, “Identifying compiler options to minimise energy consumption for embedded platforms,” *The Computer Journal*, vol. 58, 03 2013.

[13] A. Waterman, “Improving energy efficiency and reducing code size with risc-v compressed by,” 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:268101087

[14] C. Lattner and V. Adve, “Llvm: a compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[15] A. Hager-Clukas, J. Schröter, and S. Wallentowitz, “Sizalizer: Multilevel analysis framework for object size optimization,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation: 24th International Conference, SAMOS 2024, Samos, Greece, June 29 – July 4, 2024, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2025, pp. 108–121.

[16] A. Hager-Clukas, P. Kempen, and S. Wallentowitz, “Arise: Automating risc-v instruction set extension,” 08 2025.

[17] R. Gonzalez, “Xtensa: a configurable and extensible processor,” *IEEE Micro*, vol. 20, no. 2, pp. 60–70, 2000.

[18] V. Herdt, D. Große, H. M. Le, and R. Drechsler, *Extensible and Configurable RISC-V Based Virtual Prototype*. Cham: Springer International Publishing, 2020, pp. 115–134.

[19] J. Zielasko, R. Krauss, M. Merten, and R. Drechsler, “Improving virtual prototype driven hardware optimization by merging instruction sequences,” in *2024 27th International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*, 2024, pp. 73–78.

[20] Fisher, “Trace scheduling: A technique for global microcode compaction,” *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478–490, 1981.

[21] D. Patterson, J. Bennett, M. Bennett, H. Chelin, D. Harris, J. Hellar, W. Jones, K. Moron, P. Savini, R. Shepherd, R. Simar, Z. Susskind, and S. Wallentowitz, “Embench iot 2.0 and dsp 1.0: Modern embedded computing benchmarks,” *Computer*, vol. 58, no. 5, pp. 37–47, 2025.

[22] Google, “Tensorflow litert for microcontrollers,” https://ai.google.dev/edge/litert/microcontrollers/overview, 2024, accessed: 2025-09-13.

[23] C. Papon and Y. Xiao, “SpinalHDL,” https://github.com/SpinalHDL/SpinalHDL, 2016, lGPL-3.0 License.

[24] P. M. et al., “Sail risc-v model,” https://github.com/riscv/sail-riscv, 2024, gitHub repository, accessed: 2024-09-13.