Virtual Prototype driven Application Specific Hardware Optimization

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

***Abstract*—Most hardware in the area of IoT and embedded systems only ever runs a single application. To reduce the cost and increase performance the hardware can be tailored to this application. Unfortunately, identifying, designing, and evaluating application-specific optimizations is complex and requires significant effort. However, application-specific hardware also performs significantly better compared to using general-purpose processors. Prior work attempts to address this problem via approaches from the *Register-Transfer Level* (RTL) as well as the application level, with RTL being effective but resource-intensive, while high-level approaches are faster but lack accuracy.**

In order to combine the advantages of high-level and low-level approaches we propose an open source *Virtual Prototype* (VP) based workflow to automatically identify promising hardware optimization candidates based on recurring patterns. Our results demonstrate that a VP can be used effectively as a starting point for application-specific hardware optimization.

I. Introduction

In recent years, the demand for high-performance applications in the fields of IoT and embedded systems has risen significantly. However, due to power and area constraints, traditional *General-Purpose Processors* (GPPs) have become increasingly unsuitable for such applications, as most of these GPPs are only ever used to run a few or even just a single program [1]. While GPPs offer the advantage of a much lower amortized development cost (as it is used in many different applications) and also the availability of a software development ecosystem, it has a considerable overhead when only used for this single embedded application, as they are not optimized for the specific requirements. The design of *Application Specific Integrated Circuits* (ASICs) and *Application-Specific Instruction-set Processors* (ASIPs) offer a solution to this problem. ASICs are processors that are tailored to a specific application with the goal of drastically improving area utilization, power consumption and performance compared to GPPs. Unfortunately, the design of an ASIC is a complex and time-consuming process, that requires a high level of expertise and experience. To address these challenges, there has been work to aid the design and evaluation of custom processors. The two major areas are high-level compiler [2] and simulator-based approaches and low-level RTL-based [1] approaches, with most existing work focusing exclusively on one of the two. The high-level approaches are comparatively fast and easy to use and have access to high-level control flow and source code, but lack important low-level information such as accurate timing and performance estimations. This results in less accurate optimizations. Conversely, low-level approaches can analyze the application behavior on the gate level with accurate timing and power information. While the results from this approach are accurate, performing a gate-level simulation (as used in [1]) is much more complex and slow while also lacking some of the higher-level information that would allow for high-level optimizations.

In between, there exist Virtual Prototypes that model an entire system at the Electronic System Level. Virtual Prototypes combine the advantages of a high-level simulator with many of the advantages of lower-level approaches. This allows us to gather additional information about the application behavior which can be used to drastically improve the analysis step, while the accurate simulation of the system improves the relevance of the results.

In this paper, we intend to bridge the gap between the existing high-level and low-level approaches and propose an improved VP-driven approach on application-specific hardware optimization, which identifies accurate optimizations while still being efficient and easy to use. We focus on the VP-based analysis that is used to determine the most promising optimization candidates. We use the RISC-V *Instruction Set Architecture* (ISA) as a case study as it is open source, and, most importantly, designed to be modular and easy to extend. The tool we developed is based on the open source RISC-V VP [3], available at GitHub [4].

There exists an overlap between the optimizations identified by tools from the 3 different abstraction levels, which causes optimizations from higher levels to affect the optimizations identified by tools from lower levels. This does not necessarily imply that combining analysis and optimization strategies from different levels is ineffective. On the contrary, we designed our tool to be compatible with existing high and low-level techniques. To try to estimate the diminishing returns of applying high-level optimizations, we test the influence of compiler optimizations on the optimization candidates identified by our tool.

This paper makes the following contributions:

- We extend on the VP-driven approach of Design, Implementation and Evaluation of RISC-V Instruction Set Extensions.
- We present an automated methodology for identifying application-specific hardware optimizations.
- We estimate the potential for performance gain for identified optimization candidates.
- We demonstrate and evaluate our new tool based on applications representative of typical embedded system applications.

To stimulate further research on this topic, we provide our tool as open source. [1]

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within projects Scale4Edge under grant no. 16ME0127, ECXL under grant no. 01IW22002 and VE-HEP under grant no. 16KIS1342.

979-8-3503-0737-5/23/$31.00 ©2023 IEEE

[1]https://github.com/agra-uni-bremen/opt-vp

II. Related Work

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

A recent paper [5] proposes a VP-driven approach for identifying suitable application-specific RISC-V extensions. The approach observes the instructions executed in the *Instruction Set Simulator*(ISS) at runtime to identify potential optimizations. The application itself is treated as a black box, while the goal is to find an appropriate instruction sequence that can be replaced and compressed into a single instruction to reduce fetch time. The paper demonstrates the advantages and flexibility of a VP-based approach and the extendability of the RISC-V ISA. The proposed dynamic execution analysis tool shows promising results, although there is still room for improvement in handling more complicated instructions like jumps and branches and improving the estimations.

B. Instruction-level Parallelism

As the parallel execution of instructions offers a considerable performance increase, there has been a great amount of focus on the design of superscalar Out-of-Order cores.

The paper on the Load Slice Core Microarchitecture [6] goes into detail on the effects of instruction-level parallelism and proposes an approach to designing an extension for in-order, stall-on-use cores that drastically improves performance while still being energy-efficient. They perform a sophisticated data dependency analysis to identify the dependency between instructions. Their approach to identifying the dependencies in sequences of instructions is similar to the one used in our tool. However, in our case, we can simplify parts of the analysis, as the underlying RISC-V ISA, being a load-store architecture, simplifies the handling of many instructions and we ignore caches.

In more recent work, Freeflow Core [7] improves on the Load Slice Core. Freeflow Core is a new architecture that selectively exploits inherent instruction level parallelism in applications to improve performance of Out-of-Order superscalar cores. It detects instructions dependent on unresolved memory instructions and guides them to a dedicated independent execution path, enabling younger ready instructions to free flow through the compute pipeline. The proposed architecture outperforms in-order and Load Slice Core both in performance and energy efficiency metrics.

C. Combining high-level and low-level analysis

The idea of combining the advantages of high and low-level approaches was also explored by Prof5 [8]. Prof5 is a profiler for RISC-V designs, which combines functional simulation with energy and timing models calibrated from RTL simulations and power analysis. It uses the existing RISC-V simulator Spike [9] to generate execution logs, which are then analyzed to estimate the timing and energy usage for the entire execution, user, and system modes or individual functions. Profiling requires no changes to the executable and compared to a standard RTL evaluation method reaches an accuracy of 95% for timing and power estimation. As Prof5 uses an unmodified simulator and can only access the information contained in the execution log, it has some limitations. For example, it can only evaluate the core and can not cover more complex architectures or caches.

D. Gate Level Optimization and Symbolic Execution

The paper *Bespoke processors for applications with ultra-low area and power constraints* [1] proposes a low-level approach to designing application-specific bespoke processors for ultra-low area and power-constrained systems. It relies on automated gate-level symbolic simulation to identify gates that are never toggled by the target application. The identified gates are then removed to create a new bespoke processor design with significantly lower area and power consumption than the original general-purpose processor. The resulting design does not require any updates or modifications of the application. They address the problem of supporting multiple applications or updates to the original application, which can be a problem as eliminating unused gates is a destructive optimization that breaks compatibility with applications that use any of the removed gates. One proposed solution is to tailor the bespoke processor to mutations of the target application to increase the chances of remaining compatible with patched versions of the application in the future. The average area and power savings achieved by this approach are a 62% reduction in area and 50% in power which can be further improved to 65% by exploiting the timing slack introduced by the gate removal. While this approach is very well suited for reducing area and power consumption, its application for designing performance optimizations is limited.

Implementing destructive changes for bespoke processors or any other type of potentially destructive optimization requires symbolic execution, as without it we can only guarantee the correct behavior of the hardware for the unchanged application with inputs used during the analysis step. Otherwise, any untested input might reach parts of the program that still require gates that have been removed. This means if we want to use a VP-driven approach to design destructive optimizations, we also need to perform a full symbolic exploration during the analysis. While there exists a variant of the RISC-V VP that supports symbolic execution [10], we decided to use the base VP. This is because execution traces from a symbolic execution are unsuited as a base for designing performance optimizations. Using symbolic execution as a base results in optimization recommendations that are tailored to the application with no specific inputs (or normalized for all possible inputs), while in practice, specific inputs are likely to occur much more frequently than others. For this reason, the application-specific optimization analysis should design optimizations based on inputs that are likely to occur in practice so that the estimated performance gain is also likely to be reached in practice. Otherwise, the tool might for example choose to optimize a loop that is in practice and with realistic inputs only run once and then escaped, but by using symbolic execution as a base, it makes up a large amount of execution time, as for specific inputs, the loop is executed indefinitely and therefore shadows the better optimization candidates.

III. Preliminaries

A. RISC-V

RISC-V is a general-purpose Instruction Set Architecture that was originally designed for research and education purposes, but it aims to become a free and open industry standard [11]. The project began in 2010 at the University of California, Berkeley, and is now managed by the non-profit RISC-V Foundation, which was established in 2015. The RISC-V Foundation consists of industry leaders, such as Google [12] and Oracle [13], as well as academic partners, and is responsible for managing the standard, creating compliance tests, and organizing the RISC-V community.

RISC-V was developed as an alternative to the widely established x86 and ARM architectures, which have a high degree of

complexity and restrictive licensing models, making them challenging to use for academic and experimental purposes [13].

The ISA is designed to be clean, modular, and scalable, with the goal of avoiding over-architecting for specific architectures or implementations. Its minimal integer base instruction set is straightforward to implement and understand, making it well-suited for research and education. In addition to the base instruction sets and optional extensions, designers can create custom RISC-V extensions to meet specific requirements.

The RISC-V ISA uses fixed-length instructions that are naturally aligned on 32-bit boundaries [11, p.5]. It is a three-operand load-store architecture [11, p.18], which means that only designated load and store instructions can read from or modify memory. All other instructions operate solely on CPU registers. The load-store architecture simplifies tracking and analyzing data dependencies, as for other instructions, only register dependencies need to be checked. For RV32I, the registers consist of 32 32-bit general-purpose integer registers (x0-x31), along with an additional register holding the *Program Counter* (PC).

B. Virtual Prototypes

The open RISC-V ISA has rapidly gained popularity, leading to the emergence of numerous hardware implementations and high-speed Instruction Set Simulators (e.g. Spike [9]). However, in the classical design flow of a new hardware system, a significant amount of time is often wasted on designing physical prototypes, and verification of the design is only possible after it is finished. To address this issue, ISSs can be used for functional verification of RISC-V RTL implementations and early software development, saving time and shortening the time to market. Unfortunately, these ISSs are primarily designed for speed and are not easily extensible to support further system-level use cases such as design space exploration, power/timing/performance validation [3], or the analysis of complex hardware/software interactions. An industry-proven approach to this problem is the use of Virtual Prototypes in the early phases of the design flow. A VP is an executable abstract model that represents the entire target hardware platform, simulating the hardware architecture at the abstraction of the Electronic System Level. It can consist of one or multiple general or special-purpose processors as well as hardware peripherals [14]. This allows for hardware designs to be tested without the need to build or synthesize them.

The paper ”Extensible and Configurable RISC-V based Virtual Prototype” [3] proposes such a prototype for the RISC-V architecture, using the standardized C++-based modeling language SystemC [15] and Transaction Level Modeling 2.0 (TLM) to simulate hardware interactions. These TLM transactions are exchanged over a data bus architecture. This results in much faster simulation speeds compared to classic RTL and eases modeling of devices and peripherals, while still achieving a much higher accuracy than high-level ISSs. The SystemC simulator core allows for hardware modules and software to be tested on this abstract design description. This allows us to already verify software in the early development stages.

A crucial aspect of the VP is the low-level access to the system and high simulation accuracy. This access is necessary to gather all information about a program's execution during runtime that is required for identifying relevant optimization candidates. The source code is freely available [4] and written in a high-level language, which enables all modifications necessary to obtain this information. A high level of accuracy is equally important as any inaccuracies or errors in the program execution and system behavior potentially render all obtained results useless when applied to a real system without those specific inaccuracies.

IV. Methodology

In this section, we present our proposed methodology on identifying promising hardware optimization candidates using the RISC-V *Instruction Set Architecture* (ISA) as a case study.

A. Overview

We use the RISC-V VP as a base and extended the Instruction Set Simulator core with a tracing and analysis module. The ISS is the central component of the VP and handles the decoding and execution of individual instructions, thus we can get a detailed insight into the execution. The resulting *RVOPT VP* can be used in the same way as the RISC-V VP to execute RISC-V programs, but additionally outputs execution analysis results on instruction sequences that are the most promising candidates for performance optimization and, in the default configuration, also generates a dot graph visualization of the results. It is not necessary to modify the analyzed applications in any way as tracing is implemented in the core intercepting the decode and execute steps. Configuration of the tracing and analysis is exclusively done on the VP side as well. This ensures that the analysis results are authentic and applicable for the real application run on the actual hardware later. Otherwise, changes to the binary might affect execution and therefore the results.

B. Execution Tracing

For the analysis step, we require detailed information about each executed instruction. As this approach is code agnostic, we can not rely on the source code and binary DWARF info to later reconstruct parts of the information. This means we have to store this information during the execution of each step. While the concrete information we need to trace varies depending on the concrete instruction type, some base set is required for all of them. In most traditional execution tracing use cases, each execution step is traced and logged individually. The resulting trace is then output and processed independently of the simulator. An example of this is Prof5, which also uses this tracing type [8]. This approach has two major disadvantages: As the internal state of the simulator is not available when analyzing the trace file, we have to know all information required for it during execution or log everything that could be required, which is either not trivial or not practical. The second problem is also the scaling of the trace for realistic applications.

A tracing technique better suited for our approach is to create bounded execution trees. This allows for a drastically reduced trace size by removing most of the unnecessary and duplicate information for instructions that are executed multiple times. By analyzing the trees during runtime we also have access to all other runtime information from the simulator.

C. Internal Graph Structure

As mentioned in the previous section, we use a bounded execution-tree-based approach for saving all required information during runtime effectively and efficiently.

Internally a ring buffer is used to store information about the last N instructions. Once an instruction would leave the ring buffer, we insert its information and that of all subsequent instructions from

the buffer into a tree structure or create a new tree if this is the first time this instruction is encountered during execution. This way we construct a separate tree for each different instruction, always containing the corresponding instruction itself as a root node. From the root node, we start to insert new nodes for each subsequent instruction in the instruction sequence currently held inside the ring buffer or update the weight and data of nodes that were already contained in the tree. As insertion always starts at the root node, the tree can only grow up to depth N.

The number of nodes grows with each newly executed unique instruction sequence, which results in a worst-case complexity of M possible new branches at each existing node, where M is the number of instructions supported by the core. While this theoretical complexity is astronomically high, in practice the number of branches rarely reaches values above 20 for the root node and roughly halves with each level of depth as can be seen in Fig. 1. During execution, values close to the maximum tree size are reached almost immediately in most cases. This relatively low complexity and slow growth after the initial few cycles are expected, as the tree only grows if new code is executed during simulation because each new node added to the tree represents a previously unexplored part of the program.

Executing the same code/instruction sequences over and over only update the already existing nodes in the tree, meaning the size of the trace does not grow, safe for a handful of special cases.

Fig. 1 shows an excerpt of the dot graph visualization of the internal tree structure for the *ADD* instruction execution tree that is created during the execution of the *md5sum* benchmark used in the evaluation of our tool (see Section V). This execution tree is created for each different instruction, with the corresponding instruction as the root node. In this example, we can see that the *ADD* instruction was executed 639.426 times, which accounted for 27.3% of all executed instructions. From the root node, a large number of branches lead to its child nodes, which represent the next instructions that were executed after an *ADD* instruction was encountered, with the number on the edge representing the number of times this instruction sequence occurred. For example, in this specific case, for all the 639.426 times the *ADD* instruction was executed, in 56.015 cases, the next instruction was a *Load Word* (LW) instruction. The sequence following the two instructions is also shown in the zoomed-in box. This sequence continues up to a final leaf node at depth *N*. The depth is chosen arbitrarily and set before executing the program on the simulator. Larger values impact the performance of the simulation as the size of the tree and the ring buffer, as well as the number of tree operations that have to be performed, grows exponentially with *N*. However, if the depth chosen is too small, it limits the scope of the analysis step and the maximum size of identified optimization candidates. A good value for *N* is best determined experimentally as one can simply increase and rerun the simulation and analysis until the maximum length of identified optimization candidates stops growing (i.e. is less than *N*). In our experiments, very few applications required a depth greater than 50, which roughly tripled execution time compared to initial testing with smaller depths of around 20.

Depending on the specific instruction, the data that has to be stored differs, but some base set of data has to be stored for every instruction. The Information traced for each instruction is:

- The **weight**, representing the number of times this node occurred/instruction was executed.
- The **total cycles**, the simulator spent executing this instruction.
- The **powermode**, the simulator was in executing this instruction.
- The **registers**, a list of registers that were used (rs1, rs2, rs3, or rd) when this instruction was executed.
- The **data dependencies**, a list of offsets to previous instructions in the sequence on which it depended at least once when it was executed.
- The **sum of step ids**, giving a rough estimate of when this node occurred most frequently during execution (e.g. an initialization loop that was only executed at startup).
- The **subtree hash**, identifying this specific sequence of instructions.

Branch and jump instructions also save a list of **Jump Targets** the instruction jumped to as relative offsets. This information is required to properly handle loops later during the analysis step. Load and store instruction save a list of **memory accesses** and the memory region (e.g. Stack, Heap) it accessed. Lastly any leaf node at depth *N* also stores a list of **PCs** it was executed at. For higher bounds, this contains almost exclusively a single PC, but for lower bounds or applications with a high level of code duplication, this can contain multiple PCs. This also highlights how this approach is source code agnostic as it allows us to identify optimizations depending on the actually executed instructions decoupling this approach from compiler-based approaches and assuming an already perfectly optimized application binary. Some of this information is also incorporated in the visualization in Fig. 1. For example, the numbers below opcode names indicate *True dependencies* to preceding nodes in the sequence at that offset. It is important to note, that the visualization omits any branch with a weight below a certain threshold. This can be seen in branches that end before the maximum depth. Without pruning the tree in the visualization, the graph can not be rendered properly using a dot renderer, however, the missing branches are still used during the analysis, though they are unlikely to be promising optimization candidates.

During execution, whenever we encounter a sequence or partial sequence of instructions that are already contained in the tree, we update the existing nodes instead of inserting new ones. This is a very effective method of compressing the information contained in the trace without losing any data required for the analysis step.

Tracing register dependencies operates in the same bound *N* used for the execution trees, while memory dependencies are additionally traced globally using the PC of the accessing instruction. For both memory and registers, we use tainting to track the effects of writes on following instructions. For every instruction that writes to a register $\neq x0$, we taint that register with the current ring buffer index. For every following instruction in the sequence that depends on a tainted register, we add the corresponding offset to the list of dependencies. The dependency relations can later be used in the analysis step to improve the relevance of identified optimization candidates.

D. Analysis

To find the most promising instruction sequences, we analyze all execution trees we created during the simulation of the application. With the default configuration, we proceed as follows: For each tree, we start at the root node and try to find the path with the highest score. For each node type, we defined a configurable score function that returns a score value estimating how important this

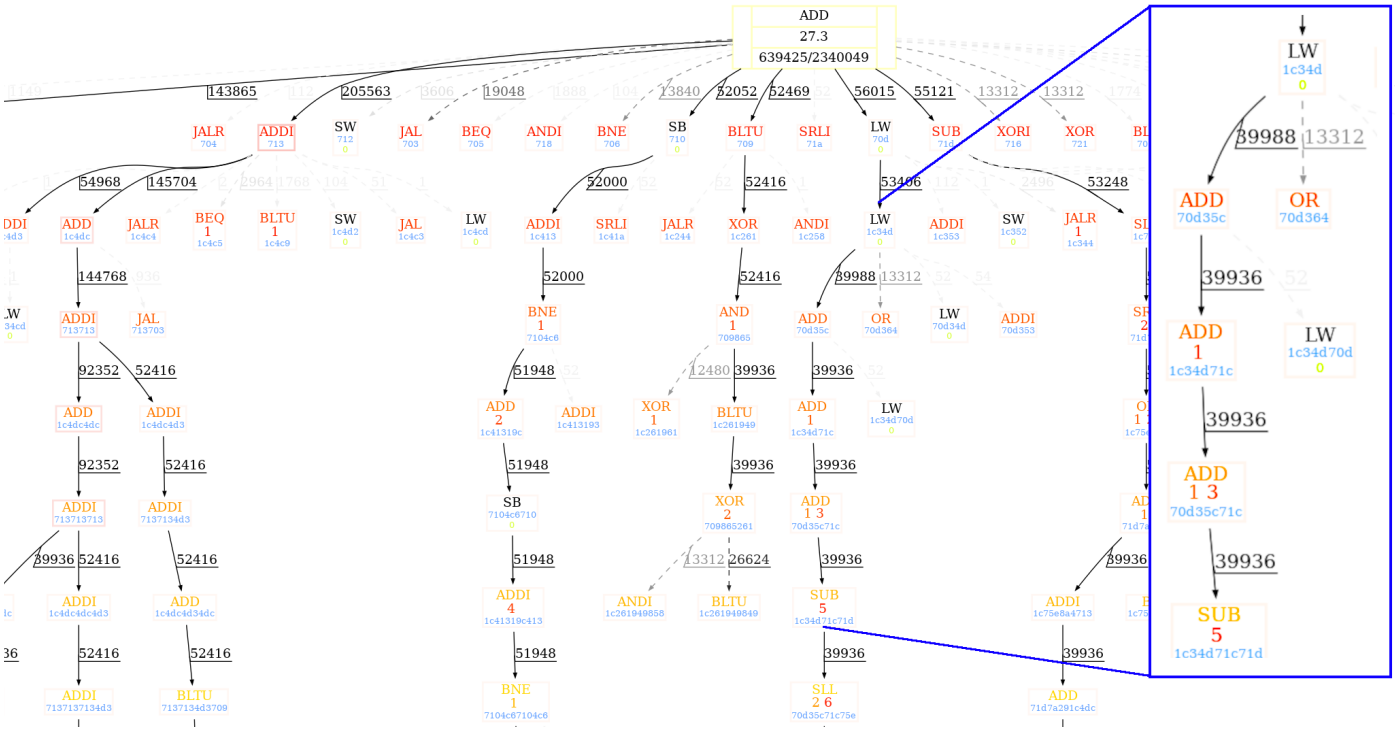

Fig. 1. Excerpt from a bounded execution tree for the ADD instruction

node is for the execution and how well it is suited for optimization. The base score value of a node is defined by the weight of the node (or the total cycles if configured), which for the root node of a tree represents the total amount of times this single instruction was executed during simulation. Now we try to extend the path from the root node to increase the total score of the path. For this, we check all branches and test how the total path score changes if we consider them part of the path. In the case of the simple score function that uses the weight of instructions, we now take the minimum weight of all instructions in the path and multiply it by the length of the path. For each instruction we add to the path, the weight can only ever decrease or stay the same. Using this function we end up with the instruction sequence making up the highest percentage of executed instructions executed by the simulator regarding sequences that start at the instruction at the root node. Sorting the identified best paths of each tree gives us a list of promising optimization candidates. Many optimization candidates identified using this simple score function are suitable for various optimization techniques, however, there are several cases that cause the analysis to miss the better optimization candidates or even result in an inflated score of certain instruction sequences. One important special case that has to be considered when analyzing a tree is instruction sequences that contain branches or jumps. Especially branches that result in a loop can be a challenge to handle correctly. This problem is also a limitation in previous work [5]. To properly handle branches and jumps, a more granular score function is required. Our approach to better incorporate the influence of individual instructions on the total score is to track a score bonus and a sequence-wide score multiplier that is adjusted by each new instruction we add to the path. The score multiplier is used for instructions that increase or, in most cases, decrease the optimization potential of the whole sequence. The most common example is an instruction we do not want to add to the sequence. In this case, setting its score multiplier to 0 prevents it from ever being chosen during path extension. In contrast, the score bonus can be used to change the value of an individual instruction. All score bonuses of a path are added together and the resulting value is multiplied by the minimum path weight and subtracted from the total score. An example of this is an instruction that does not provide any direct optimization potential, but a path extending past it might still increase the total potential. In this case, setting the bonus to -1 causes the instruction to be essentially ignored when calculating the score during path extension. We have to track the bonus score during the analysis and can not simply subtract the weight of the current instruction from the score in this case, as the minimum weight might still change when new instructions are added to the path. This addition to the score function allows us to properly handle jumps and branches, while allowing an easy way to tweak the score function to identify optimization candidates more suitable for specific hardware optimization techniques. For example, adding a score bonus for instructions with few and a penalty for instructions with many data dependencies will results in sequences that are easier to use as a base for parallelizing or reordering instructions.

Conditional Branches pose an additional challenge during analysis and, unless we want to design an application-specific branch prediction backend (see Section IV-F), are not promising inclusions for our optimization sequences. Generally, there exist 4 types of branch behavior we have to consider: First is the case in which the branch is not taken and acts as a *NOP*. In this case, we do not have to do anything special besides assigning a score value to this branch type. By default, we assign it a score bonus of -1. In the second case, the branch jumps to its target address, which is unrelated to the previous instructions in the sequence. In this case, we also assign it a bonus of -1. In the 3rd case, the branch loops back and includes the root node in its loop. This proper loop creates the problem of nodes in the sequence being counted multiple times when extending the path. In this case, we have to either terminate

the path when we encounter this type of branch or adjust the weight to account for counting the sequence twice. The former can be done by setting the score multiplier to 0, which results in the branch instruction never being chosen during path extension. In most cases for loops, this results in the path ending at this node as any other chosen node would leave the loop after its first execution, likely resulting in a much lower minimum weight. The latter requires a proper analysis of the loop, but allows us to extend the sequence over multiple iterations, essentially unrolling the loop. In the last branching case, the branch is a loop that branches back, but does not include the root of the tree, but at least one other instruction in the sequence. In this case, we can include it without running into the aforementioned problem by setting its score bonus to -1 identical to the other 2 cases. This leaves us with a general score function for our analysis that can identify promising optimization candidates that are suitable for a wide range of hardware optimizations. We use this analysis approach to identify the most promising sequence for each execution tree. The resulting list is sorted by their score value and output with some additional general information about the analysis and execution.

E. Optimization Potential Estimation

Using the instruction sequences identified in the analysis step, we can estimate, how much potential for application-specific optimizations each sequence holds. To compare the results of the analysis for different inputs or applications, we need a normalized value. For this reason, we introduce the *Normalized optimization Potential* (NP). This value is calculated by:

$$\frac{Length \times Weight}{Total\_Instr \times \sum_{node=1}^{Length} \frac{1}{dep\_offset_{node}}}$$

Where the inverse dependency score on the bottom right is calculated based on the data dependencies between nodes in the sequence. Specifically, we iterate over each node in the sequence, and for each dependency, add the value $1/dep\_offset$ to the inverse dependency score. The exception is a sequence of length 1 for which this potential is defined as 1. This results in a low inverse dependency score for sequences with few direct data dependencies on preceding instructions, which offer more opportunities to reorder instructions in the sequence.

F. Designing and Testing Hardware Optimizations

The analysis results give a detailed view of the execution and suggest the most promising optimization candidates. The tool does not however automatically design a concrete hardware optimization based on the best results. For the design of concrete hardware optimizations, we can, for example, follow the existing VP-driven approach [5] and design fused instructions that reduce fetch time.

To test new instructions, it is normally necessary to modify the simulator as well as the compiler that generates the binaries to include the new instructions. Using the results from our analysis, however, we can simulate and test new instructions without having to modify the compiler or change the binary in any way. The path hash that is calculated for each instruction sequence uniquely identifies that sequence. By using the same hash function during simulation to calculate the hash of the next-to-be-executed instruction, we can identify every occurrence of the sequence during simulation. Whenever this happens, we can discard the effects of the original sequence and instead, simulate and test the effects of the new instruction or hardware optimization we designed for the sequence. This naive approach can become complicated for larger instruction sequences, as we can only identify the sequence with certainty at the last instruction in the sequence. At this point discarding and replacing the effects would require to undo all the previous steps in the sequence. There are two ways to solve this problem. First, we could prefetch the next *n* instructions, where *n* is equal to the length of the sequence. This approach works well for most sequences but fails if the sequence contains any conditional branches. Alternatively, we can rerun the tool by additionally specifying the path hash as an argument. During the simulation, the tool also collects any sequence of instructions with length *p* preceding the target sequence and calculates the corresponding path hash. The output is a list of path hashes for which we can be certain, that whenever they are encountered, the next executed instructions are the target sequence. The problem is, that we have to discard any preceding sequence that can also leads to a different sequence. In practice, however, this approach should be accurate enough to estimate the performance gain of the optimization, as during our experiments, with a *p* of more than 20, this problem did not occur. Especially if the program is run on the same inputs as those used for the analysis. Further optimizations are pipelining, multiple-instruction-issue or caching as we have detailed statistics for data dependencies between instructions in a sequence and global memory access patterns.

Another use case for the execution trees is the design of a branch prediction backend. When looking at branch instructions contained inside a tree, we can calculate the likelihood of whether the branch is taken or not by comparing the weight of outgoing branches. During execution, whenever the previously executed instructions match this sequence from the branch node up to the root node, we can simply look up the corresponding value. A good starting point for developing a brach prediction backend is the *Berkeley Out-of-Order Machine* (BOOM) [16], which already implements two levels of branch prediction for a RISC-V core. Designing an improved branch prediction backend using the analysis results is currently planned for future work.

V. Evaluation

To evaluate our VP-based approach, we used it to analyze Embench [17] binaries and two programs running inside the RIOT [18] OS. Embench is a free and open-source benchmark suite designed to test the performance of deeply embedded systems providing a broad range of different applications that are representative of typical applications in the domain. This allows us to evaluate how our analysis approach performs for different types of applications and the underlying algorithms. It is easy to use and used in many other projects for evaluation purposes, which can make comparison between different approaches easier. To cover the use case of hardware running an operating system, we also evaluate our approach using RIOT. Similar to Embench, RIOT is open source and targets low-end embedded devices, specifically in the area of IoT.

The goal of this work is to bridge the gap between high and low-level approaches. To this end, we seek to understand the compatibility between and the effect of high-level optimizations on low-level ones. One of the most commonly applied high-level optimizations is compiler-based code optimization. To understand the effect of high-level compiler optimization on discovered hardware optimizations, we analyze all applications twice: Once with full compiler optimizations turned on (*-O3*) and once without any compiler optimizations (*-O0*). For the analysis, we use a depth of 50 and the

TABLE I
ANALYSIS RESULTS

Application	O0					O3				
	Root	Len	Weight	Total # / %	NP	Root	Len	Weight	Total # / %	NP
aha-mont64	BGEU/LW	38	162816	13870K[44.60%]	78.68	SRLI	11	162432	4532K[39.4%]	185.0
crc32	ADDI	19	175104	6660K[49.94%]	86.27	JAL/LW	12	175104	3846K[54.62%]	71.5
edn	BGE/LW	23	220000	12647K[40.02%]	40.0	LH	5	290400	3483K[41.68%]	104.2
huffbench	BLTU/LW	33	67692	7850K[28.4%]	38.8	ADDI	1	661440	2515K[26.29%]	1
matmult-int	ADD	48	376000	18908K[95.4%]	119.3	ADD	5	357200	4426K[40.35%]	80.7
md5sum	LW	41	53248	4430K[49.2%]	69.9	SLLI	24	39936	2339K[40.9%]	105.3
minver	ADDI	1	1070812	7375K[14.5%]	1	SW	5	100114	2818K[17.7%]	88.8
nettle-aes	LW	47	98592	7612K[60.9%]	80.2	LW	31	32864	4481K[22.7%]	99.7
nettle-sha256	ADD	50	56168	6275K[44.75%]	60	XOR	1	670685	3973K[16.88%]	1
nsichneu	ADDI	2	916624	3859K[47.5%]	95.0	LW	1	1227109	2238K[54.8%]	1
picojpeg	SLLI	24	100352	12977K[18.56%]	31.8	ADDI	1	632637	4198K[15.07%]	1
primecount	LW	6	1362394	14492K[56.4%]	84.6	ADDI	2	699732	4290K[32.6%]	65.2
qrduino	LBU	1	2291333	7807K[29.34%]	1	ADD	1	619975	3427K[18.09%]	1
sglib-combined	LW	1	2653499	6631K[40.01%]	1	LW	1	489436	2414K[20.27%]	1
slre	LW	1	1930888	6129K[31.5%]	1	SW-SW	7	107007	2570K[29.145%]	204.0
tarfind	ADDI	22	36960	2048K[39.7%]	64.7	JAL/LW	12	36960	1005K[44.12%]	57.76
ud	LUI	2	149444	10624K[33.5%]	41.2	LW-LW	2	149444	915K[32.66%]	65.31
RIOT Hello World	ADDI	1	4075	15K[26.99%]	1	ADDI	1	3893	13K[28.06%]	1
RIOT default	ADDI	1	3556	11K[29.69%]	1	ADDI	1	3370	10K[30.89%]	1
Average	-	19	1234690	41.1%	47.18	-	6.52	339986	30.95%	59.76

weight-based score function proposed in Section IV-D as it demonstrates a general use case. For designing specific optimizations, the score function can and should be adjusted to improve the practical relevance of identified optimization candidates. Table I shows the statistics for the most promising optimization candidate discovered for each analyzed application. Each row lists the application and its corresponding information on the most promising sequence that was discovered. The two main columns compare the results for the two different compiler optimization levels. The *Root* column lists the first instruction of the sequence (or the first two in case of jumps and branches). After it follows the *Length* of the sequence, its *Weight*, i.e. the total number of times this sequence was executed, and after that, the total number of instructions executed to give a frame of reference of how much of the total execution time the sequence amounts to. The last column lists the *Normalized Optimization Potential* (see Section IV-E), which represents the relative optimization potential of the sequence. For the binaries with no high-level optimizations, we were able to find promising optimization candidates for most applications. The most interesting results are for the applications *matmult-int*, *crc32*, and *nsichneu*. All of them start with an ADD/ADDI instruction, have a high NP, and make up a large part of the total execution time. The sequence of 48 instructions identified for *matmult-int* for example makes up 95,4% of execution time, being executed 376.000 times. The sequence for the application *nsichneu* in contrast only contains 2 instructions which make up 47,5% of execution time with an equally high NP, resulting from the two instructions having no data dependency. The result for *nettle-aes* is also very promising although as it has a length equal to the bounded tree depth, the results could potentially be improved by increasing the depth during the analysis. For some instructions, the identified sequence is a single instruction. This is usually the case for applications that are harder to optimize. A good example are the applications running inside RIOT. For these examples, we only execute a tiny amount of instructions compared to the other examples, which are rarely spent executing the same code. In contrast, for the *edn* application, we likely identified part of a loop that executes 220000 times, which makes it an easy target for optimizations. Applications with single instructions sequences as their best optimization candidate do not necessarily mean, that we could not identify actual instruction sequences. In all cases, looking at the list of optimization candidates there were at least two sequences longer than 3 instructions among the best 5 sequences for each application. The average length of the most promising instruction sequence is 19 instructions, which in most cases is a loop or at least part of a recurring function. The average NP is 47.18, giving a reference value to compare applications to, to get an estimate of how much potential they hold for optimization.

Comparing the results for the application without optimization to those from the optimized ones, we can identify general relations between them. For every application, the sequence length has at least halved with an average length of about $1/3$. This is expected, as compiler optimizations reduce code size and remove unnecessary instructions. The weight of the sequences shows a similar reduction, but the percentage of total execution time is surprisingly almost identical for most applications. One aspect that is hard to automatically determine or reflect in this table is the relation between the two instruction sequences identified for an application. The best sequence for *O3* is not necessarily the optimized variant of the *O0* one. However, it is possible to identify the corresponding sequence in the two full lists manually. One interesting observation from the results table is that optimized sequences can be identified quite easily with high accuracy for some of the applications. For cases in which the weight of both sides is equal or nearly equal, the *O3* side lists the optimized equivalent in almost all cases. In

many other cases, however, the identified sequence is a single instruction. This means, that the optimization can make certain identified hardware optimizations obsolete or at least more difficult. Contrary to our expectation, the NP increased on average. For some applications that have a proper instruction sequence for the *O3* case, the optimization potential more than doubled, while the execution percentage and Weight stayed roughly the same. This is likely the result of compiler optimizations removing data dependencies between instructions or at least reordering instructions.

Overall the evaluation shows promising results for the use of VPs in the design flow of hardware optimization. Our results indicate, that, while there is a level of redundancy when combining high and lower-level approaches, it can improve the efficiency of the lower-level tool for certain applications.

VI. Conclusion

We have presented an approach for identifying suitable application specific hardware optimization candidates based on a Virtual Prototype that bridges the gap between existing high-level and low-level approaches.

By combining the advantages of both, we can recommend optimization candidates that span a larger amount of instructions while being better tailored to the specific application with a high optimization potential. Furthermore, we have presented an implementation of our proposed approach in the form of an extension for the RISC-V VP [4]. In order to evaluate our approach, we have identified the best optimization candidates for the benchmark suite Embench [17] and examples running on the operating system RIOT [18]. To understand the effect of high-level compiler optimizations on the optimization candidates, we also performed the evaluation with different optimization levels. The results of our analysis indicate, that, while the identified instruction patterns, sequence sizes, and estimated performance gain varies greatly depending on the application, we were able to identify promising optimization candidates for almost all analyzed applications with an average sequence runtime of over 41% (Table I) and an average estimated normalized performance optimization potential of 47. Combining our approach with high-level compiler optimizations showed that, while the average length of instruction sequences was reduced to less than 1/3, the optimization potential stayed the same and even increased for some applications. However, it also showed that there are many cases in which compiler optimizations solve the same optimization candidates, that were previously identified by our tool. Our approach extends on the previous work on "Virtual Prototype driven Design, Implementation and Evaluation of RISC-V Instruction Set Extensions" [5] that proposed a VP-based optimization approach and improves on its execution analysis approach, to fully leverage the advantages of a VP-based approach and enable more complex optimization recommendations and a higher potential performance gain. To the best of our knowledge, we have presented the first approach for identifying application-specific hardware optimizations candidates, that combines a VP-driven approach with the proposed execution tree-based tracing and analysis to leverage the advantages of a VP. To stimulate further research on this VP-driven approach, we have released our RVOPT VP as open-source software. We plan to improve our proposed approach in future work by improving the timing model of the VP to enable a more accurate performance estimation. Additionally, we plan to implement an optimization recommendation extension, so that the tool can automatically identify, recommend and evaluate concrete hardware optimizations. Branch prediction would also be an interesting research direction for our proposed analysis tool, as it should be possible to design application specific branch prediction engines with very high accuracy using the branching information from the execution trees.

References

[1] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori, "Bespoke processors for applications with ultra-low area and power constraints," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 41–54.

[2] 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*, 2010, pp. 000 535–000 539.

[3] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable risc-v based virtual prototype," in *2018 Forum on Specification Design Languages (FDL)*, 2018, pp. 5–16.

[4] A. U. Bremen, "RISC-V based Virtual Prototype," https://github.com/agra-uni-bremen/riscv-vp, 2018, accessed on June 19, 2023.

[5] M. Funck, V. Herdt, and R. Drechsler, "Virtual prototype driven design, implementation and evaluation of risc-v instruction set extensions," in *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2022, pp. 14–19.

[6] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 272–284. [Online]. Available: https://doi.org/10.1145/2749469.2750407

[7] R. K. Choudhary, N. Singh, H. Nair, R. Rawat, and V. Singh, "Freeflow core: Enhancing performance of in-order cores with energy efficiency," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 2019, pp. 702–705.

[8] J. Silveira, L. Castro, V. Araújo, R. Zeli, D. Lazari, M. Guedes, R. Azevedo, and L. Wanner, "Prof5: A risc-v profiler tool," in *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2022, pp. 201–210.

[9] "Spike RISC-V ISA simulator," https://github.com/riscv/riscv-isa-sim, accessed on June 19, 2023.

[10] S. Tempel, V. Herdt, and R. Drechsler, "Symex-vp: An open source virtual prototype for os-agnostic concolic testing of iot firmware," in *Journal of Systems Architecture*, vol. 126, 2022, p. 102456.

[11] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual*, 2nd ed., https://riscv.org/specifications/, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 5 2018.

[12] "RISC-V Foundation," https://riscv.org/members/, 2015, accessed on June 19, 2023.

[13] D. Kanter, "RISC-V OFFERS SIMPLE, MODULAR ISA," *Microprocessor Report*, 2016. [Online]. Available: https://riscv.org/wp-content/uploads/2016/04/RISC-V-Offers-Simple-Modular-ISA.pdf

[14] R. D. Vladimir Herdt, Daniel Große, *Enhanced Virtual Prototyping*, 1st ed. Springer, 2021.

[15] IEEE, *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.

[16] "The Next-Line Predictor (NLP)," https://docs.boom-core.org/en/latest/sections/branch-prediction/nl-predictor.html, accessed on 2023-05-10.

[17] "Embench™: Open benchmarks for embedded platforms," https://github.com/embench/embench-iot, accessed on June 19, 2023.

[18] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Riot: An open source operating system for low-end embedded devices in the iot," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.