Journal of Signal Processing Systems
https://doi.org/10.1007/s11265-023-01841-3

AEx: Automated High-Level Synthesis of Compiler Programmable Co-Processors

Alex Hirvonen[1] · Topi Leppänen[1] · Kari Hepola[1] · Joonas Multanen[1] · Joost Hoozemans[2] · Pekka Jääskeläinen[1]

Received: 1 April 2021 / Revised: 5 April 2022 / Accepted: 16 January 2023

Abstract
Modern High Level Synthesis (HLS) tools succeed well in their engineering productivity goal, but still require toolset and target technology specific modifications to the source code to guide the process towards an efficient implementation. Furthermore, their end result is a fixed function accelerator with limited field and runtime flexibility. In this paper we describe the status of AEx, a novel work-in-progress HLS tool developed in the FitOptiVis ECSEL JU project. AEx is based on automated exploration of architectures using a flexible and lightweight parallel co-processor template. We compare its current performance in CHStone C-language benchmarks to the state of the art FPGA HLS tool Vitis, provide ASIC implementation numbers, and identify the main remaining toolset features that are expected to dramatically further improve the performance. The potential is explored with a hand-optimized case study that shows only 1.64x performance slowdown with the programmable co-processor in comparison to the fixed function Vitis HLS result.

Keywords Programmable accelerator overlay · Design space exploration · High-level synthesis · ASIP · Transport triggered architecture

1 Introduction

The usage of FPGA devices as accelerators has increased in the last decade, thanks to their flexibility. Although diversity of options has a positive impact in the computing domain, a key problem still remains: the complexity and platform-specificity of the tools required to develop useful FPGA-based designs [1, 2]. FPGAs provide a high degree of flexibility which comes with costs that need to be surpassed by means of datapath specialization, enhanced on-chip communication or additional parallelization, requiring more effort on the implementation.

Customized soft-core based overlay architectures introduce an additional software programmable layer to the FPGA-based implementation. Using the instruction-set architecture (ISA) layer, the same application description can be retargeted to different FPGA platforms implementing the overlay simply by recompiling the software. However, all overlays incur overheads. When compared to a fixed function implementation, the additional overhead of an ISA-based overlay eventually results from the instruction stream support resources required.

Further benefits of customization can be reached with new ASIC chip designs where processors and accelerators can be integrated to a System-on-a-Chip (SoC) each with desired degree of reprogrammability. However, to motivate new SoC designs, the fundamental requirement is to reach enough benefits via specialization or ownership of a new SoC IP. The benefits must overcome the high costs of new chip runs, where non-recurring engineering (NRE) costs required to design and validate the new SoC design play a significant role.

Application-Specific Instruction-Set (ASIP) processors aim to reduce the NRE costs by means of a software programmable template from which the accelerators are defined. The end result can be field-programmed by switching the software, enhancing design reuse and reducing design time validation risks thanks to the post-manufacture bug fixing capabilities.

✉ Topi Leppänen
topi.leppanen@tuni.fi

[1] Faculty of Information Technology and Communication Sciences, Tampere University, Tampere, Finland

[2] Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Delft, The Netherlands

Published online: 15 February 2023

High level synthesis tools allow [2, 3] developers to automate the generation of the low-level hardware descriptions from the application code written in high level language. Using higher levels of abstractions removes the entrance barrier for software engineers allowing them to describe the algorithms written in high-level language (HLL) without concerning themselves about the underlying hardware specifications. Ideally, such development paradigm should increase the productivity, decrease NRE costs and shorten final product's time to-market. But, compared to the hand-written register transfer level (RTL) designs, there is still a performance gap in various identified application domains [4]. In addition, many HLS tools still require developer to make modifications to the original HLL code providing pragmas or hints about the resulting target hardware, making HLL code target tool or device specific. As the developer has to still know the additional information about the target hardware platforms, the learning curve and productivity suffers despite of the automation benefits of HLS.

In this paper we describe a new HLS tool which produces an ASIP as its output, which means it is suitable both as an FPGA overlay architecture as well as for integrating into new chip designs. The tool called AEx utilizes transport triggered architecture (TTA) as its processor template, forming the basis for modular automated co-processor architecture exploration.

In our previous work we showed that specialization of the co-processors can be automated using TTAs to generate efficient end results when compared to hand-made general purpose designs [5]. This paper describes the current status of AEx in the end of the FitOptiVis project [6]. It is an extension of our previous work [5] providing these new main contributions:

- Simplified user input requirements by automated selection of resource unit amounts and threshold parameters
- Advanced heuristics for wider design space exploration and pruning
- Option to input a starting exploration architecture skeleton for predefined memories and custom operations
- Comparison to the modern HLS tool Vitis HLS by Xilinx

This paper has the following structure: Section 2 introduces the TTA template which is used in this paper. Section 3 describes the design exploration problems and introduces the proposed design space exploration heuristics. Section 4 presents the results obtained with the current status of the tool. Section 5 investigates the bottlenecks in comparison to a modern HLS tool Vitis by Xilinx via a hand-optimized case study. Section 6 walks through the most relevant related work. Finally, Section 7 concludes the paper.

2 Transport-Triggered Architectures

Transport-triggered architectures is a class of exposed datapath processors. TTAs differ from the traditional "operation-triggered processors" by having more low-level control of data transfers between processor functional units. The interconnection network of TTAs, containing function units (FU) and register files (RF) of different widths connected with buses, is exposed to the programmer. This allows it to have a simple hardware circuit with a cost of increased compiler complexity to explicitly express the instruction-level parallelism. Simpler hardware allows higher operating frequencies and lower energy consumption, which is a very important design goal for the modern mobile world [7]. Architecture resources are visible to the compiler allowing it to schedule the code efficiently utilizing all the available parallel units and the connectivity between them.

The datapath of TTA is programmed as data transports or moves between the resources, e.g. function units and register files [8]. The program consists of instruction moves between input and output ports of architecture units. The operands of the operations are transported to the specific function units and the actual operation execution is performed as a side-effect when the data is written to the triggering input port of the FU. The compiler's task is to schedule the data movement in and out of the FUs and RFs. Multiple moves can be bundled into one instruction allowing multiple parallel data transports in a single instruction. Figure 1 shows an example TTA processor containing five transport buses, which means five moves can be made in parallel. This is similar to the very large instruction word (VLIW) architectures, but the TTA does not have the RF complexity bottleneck, as its RFs can have less ports due to the exposed datapath, while providing the same level of efficiency [9].

In contrast to typical operation-triggered format, the move structure in TTAs is more flexible and its operands do not have to reside in registers. Output values from function units can be bypassed directly from function unit output ports and routed to the required port, without accessing RFs. This reduces the RF accesses which helps to minimize the RF port count, thus providing better energy efficiency and shortening the possible critical path caused by the RF with large number of ports. On top of that, the compiler has the additional freedom to control the timing of the function unit operand and result data transports. Residing registers in FU ports can also store operands and results, giving the compiler more freedom while scheduling operations. One disadvantage of increased compiler responsibility in TTAs is the additional control bits in instruction word, inflating the program image size. The

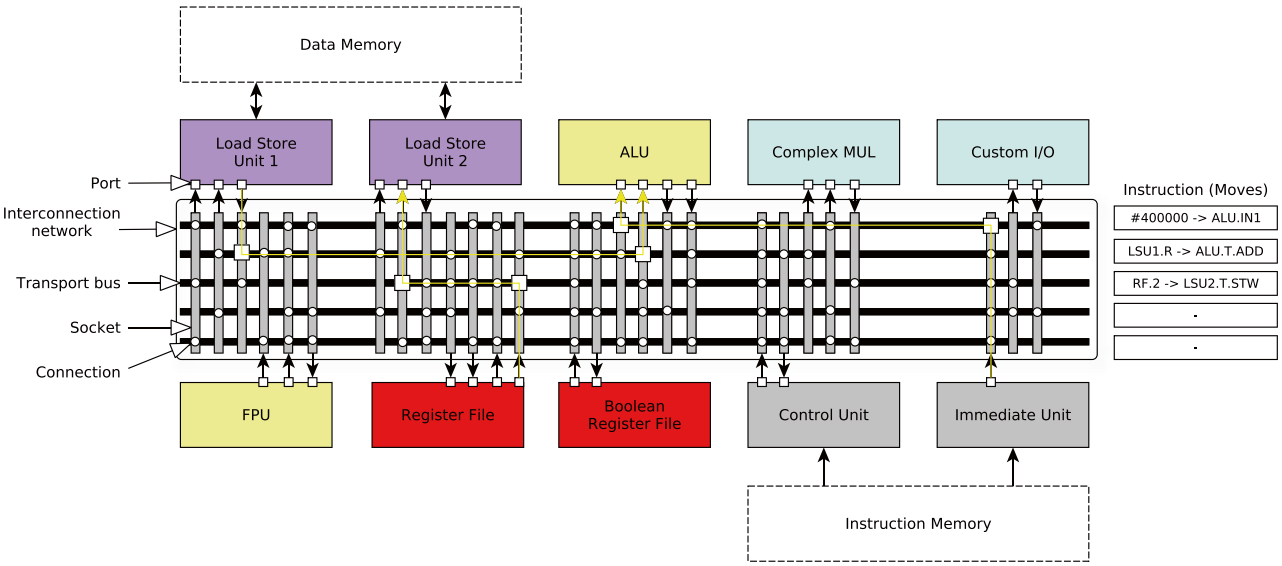


Figure 1 Example of a TTA processor. In TTA, data transports between components are explicitly programmed. Multiple function units are connected to the designated register files via input and output ports. Five transport buses make it possible to execute 5 parallel moves in a single instruction. The example instruction defines an instruction with three moves controlling data transports in three buses out of the five. The instruction performs an integer summation of a value loaded from a data memory with a constant while simultaneously storing a previously computed value to memory.

effect of this can be reduced with application-specific tailoring of the instruction set and instruction compression.

The modularity of TTAs makes it possible to easily customize a processor by adding any hardware resources to the architecture, including FUs, RFs, transport buses and arbitrary connections between these. For example, frequently used operations can be duplicated or custom operations for heavy parts of the application can be added to provide additional performance. The units can be connected together in arbitrary ways and it is the compiler's task to route the data transports efficiently. Unless the connections are carefully considered, routing difficulties can lead to the increased critical path length.

With these aspects in mind, TTAs are interesting for automated FPGA overlay and ASIP generation use due to their simplified register files, modularity and fine-grained compiler targetable structure [10].

3 TTA Design Space Exploration

Designing a processor starts with the specification of the requirements. Such requirements usually set restrictions on the available resources and execution time of the application algorithm, limiting the size, performance or energy consumption. Processor design is typically an iterative process of continuous architecture modification and evaluation, until the requirements are satisfied. A typical evaluation process consists of compiling and simulating the architecture and making appropriate changes based on the simulated behaviour of the processor. The TTA-specific architecture description language (ADL) is used to express the structure and resources of the processor as an architecture description file (ADF). This file can be used as an input to both the retargetable compiler and to the generation of the processor's HDL-description. Manually designing a processor this way is error-prone and time-consuming, and should be automated by the toolset to support faster development.

The flexibility of TTAs gives a plethora of different ways of customizing the processor for specific needs of applications. Multiple function units (FU) can be added and connected together, with each having different operations, parameters and memory structure. Widths, depths and port counts of register files can be adjusted. The connectivity between the FUs and RFs can be edited to support certain bypasses and eliminate others. The amount of parallelism in transport buses can be adjusted to find the balance between sharing of the bus and parallel data transfers. Short immediate values of different bit widths can be embedded into the instruction word. A collection of these different design parameters is called a processor configuration. All possible combinations of these parameters create a huge design space and exploring it would take a long time if each reasonable architecture configuration was to be evaluated individually. Typically, the Pareto configuration points are the most interesting design configuration points, which provide optimal results within given requirements. Finding

those points is considered to be a multi-objective optimization problem [11, 12].

As the design space is too large, a proper heuristics to shrink it must be used. Some works rely on the usage of evolutionary genetic algorithms, where configuration candidates evolve in next iteration cycles based on specific fitness score, but they do not always guarantee that the found solution is the best possible [11, 13]. In our work we explore the design space as a tree, which is pruned using specific input parameters described later in the paper.

A good exploration starting point for TTA is to create a complex VLIW-like connected architecture, where each FU port has a separate corresponding RF port, and start pruning the underutilized components, transforming it into a simpler architecture [14]. In our work the design space is gradually narrowed by transforming the huge architecture through multiple optimization passes, where the same types of resources are pruned or merged into one. This continues until the (lower boundary) performance threshold is reached.

In our proposed HLS flow the exploration starts from 1) an input application written in C, OpenCL C or LLVM intermediate representation and 2) a set of design space limiting parameters. The main parameters are the following: a) clock cycle and frequency thresholds b) desired number of feasible output architectures. There are also multiple other optimization pass specific parameters and they are used mostly for fine-tuning the end result. These include number of parallel memory function units, starting skeleton architecture with predefined components and size of the immediates. The output of the flow is the architecture description files of conforming configurations.

3.1 Exploration Pipeline

The automated exploration pipeline consists of several architecture processing passes which refine the architecture for the next optimization pass. The general idea is to perform each optimization pass in a cyclic manner starting from the compilation of an input application against a given architecture, simulating and refining it until certain limitations are reached as depicted in Fig. 2. In our work we use the expand and prune approach. In the initial stages we start from a massive architecture having maximum resources and connectivity. At each step, based on the profiling results, we start pruning the least utilized resources and merging components based on their parallel activity. Each optimization pass can produce from one to multiple architecture configurations in order of best performance in terms of cycle counts. The passes that create multiple output configurations need to be carefully joined together. New possible configurations expand as a tree structure, starting from the large starting point architecture and expanding every time there's a pass with multiple output configurations. The heuristics

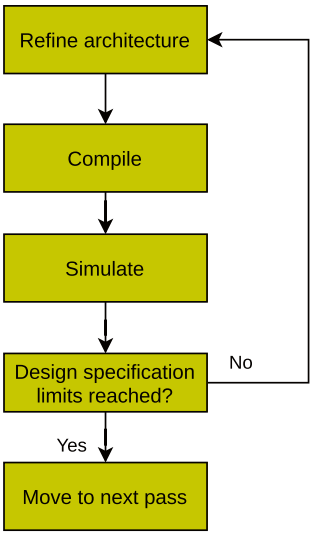


Figure 2 Iterative process of a typical exploration stage. The architecture is refined by modifying the architecture resources. The application HLL code for the produced architecture is compiled and simulated. After that, the decision is made against requirements specified by designer, to either continue modifications or finish refining and pass it to the next optimization pass. Usually the iteration continues until a certain performance or resource utilization goal is reached.

handle the selection of these architecture options and walk forward and back through the exploration tree, trying different feasible resource combinations. The overall structure of the exploration pipeline passes is shown in Fig. 3.

The large initial architecture containing the maximum set of resources is not meant to be a synthesizable processor, but is used as a starting point of the next optimization pass for the profiling and pruning of the resources. For each operation with known behavior from the operation database, an individual function unit is added with the register file of corresponding bit width. For the arithmetical and logic operations, ALUs are created, as well as load-store units (LSU) for the memory operations connected to the default address space. If a special skeleton parameter is given, the starting architecture can contain predefined ALUs and LSUs with designer defined custom latencies and address spaces. The initial size of the RFs is set high enough to store the most variables of the application to avoid spilling content to the main memory.

A starting skeleton architecture can be used to specify different external interfaces (e.g. memory). The exploration can be run using a custom starting architecture containing some function units with special operations, predefined delays and address spaces. While creating the initial huge architecture, other function units for operations are simply added to this predefined architecture, without re-adding the operations found in predefined units. Predefined function units are also kept untouched during the following optimization passes.

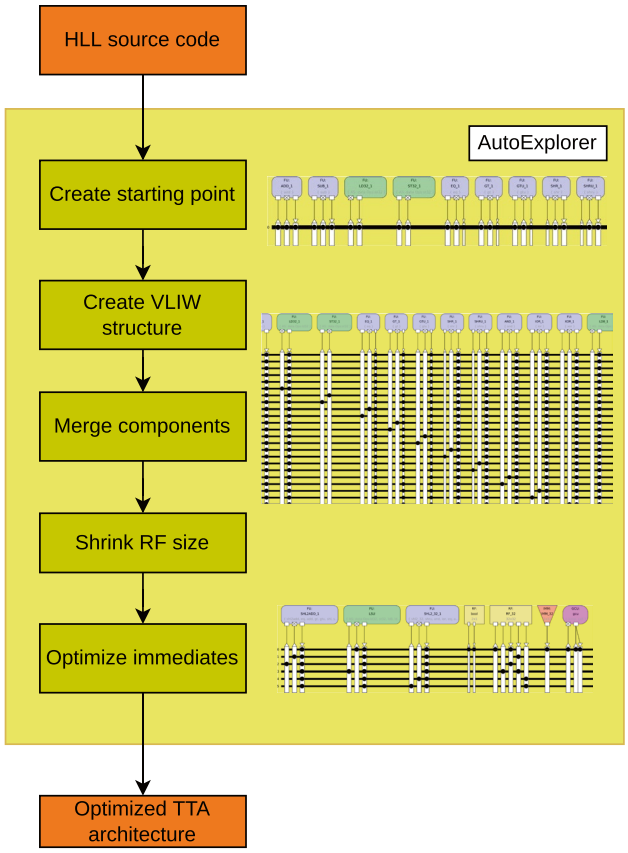


Figure 3 Automated exploration and its passes. The automated exploration of our approach starts from the input application code written in high level language, creating an oversized initial architecture, pruning the operation set, creating oversized VLIW-connectivity, pruning and merging resources based on their parallel utilization and optimizing other component features until crossing a certain lower boundary for performance. Snapshots on the right visualize the architecture's state in some of the stages; starting point, huge VLIW-connected architecture and the final optimized architecture.

The next optimization pass objective is to get rid of the FUs with operations that are not utilized enough. The program is is compiled for the gigantic architecture from the previous step and from the scheduled code the operation usage is gathered. The FUs for the operations not found in the scheduled code are pruned off from the architecture. Using simulation data the architecture can be reduced even further by removing some operations which are utilized very little and can be emulated in software (e.g. division, if it's rarely used). Remaining function units are duplicated as a crude way to get more instruction level parallelism. To retain the successful compilation in the next passes, a certain set of operations is never pruned. Such operations include addition, subtraction, shift and basic memory operations. Pruning FUs might also result in removing RFs, buses and sockets which are no longer used.

After pruning the operation set, the VLIW-like connectivity is created. In the resulting architecture every FU operand port has an individual bus connection to a dedicated RF port. This interconnection network is unnecessarily complex for the TTA programming model, but it will be simplified in the next optimization passes. A typical VLIW-connected architecture at this point has around 20 function units and 60-80 individual buses connecting FU ports to the RF ports.

Then the exploration moves on to the branching exploration passes. Two resources can be merged together based on their simultaneous usage. Such resources are FUs, buses and register files. The utilization data from simulation can be used to build the co-variance matrix of parallel usage of each component. Two components with minimal co-variance indicate that they can be successfully merged together without a significant impact on the performance. The idea behind the co-variance matrix is explained in more detail in [15]. Minimizing the number of function units also simplifies the interconnection network, as the arbitration between the operations is moved inside the function units. As the final branching exploration pass, the sizes of the large RFs are shrunk until it starts affecting the cycle count. This happens when the size of the registers is not enough to hold the most-used application variables and they are spilled to the main memory.

As the final optimizations, two immediate passes are performed that only produce a single output configuration each. The widths of the short immediate on each bus were set to 32 bits at the beginning of the exploration. At this point they can be reduced based on the profiling data from simulation, while still ensuring that the maximum number of the immediate moves can be done using short immediates. Long immediates are optimized by creating a long immediate instruction template split across several buses. Again, the simulation data is used to discover the least used buses and use them to pass long immediates. The current immediate optimizations are assumed to be good enough to not necessitate the use of branching passes.

3.2 Exploration Heuristics

At the beginning of the exploration the designer doesn't know how much resources the resulting architecture needs for the input application. For the efficient automated exploration tools, a certain limiting requirements goals for area, performance or energy consumption must be specified. In the final outcome designer could have a variety of different design configuration options to choose from, e.g. architectures with small resource usage and low-power usage or performance-oriented architecture with more resources. Using constant threshold limits to pick the single specific Pareto point configuration is fast in terms of exploration time, but not an optimal solution, as the more promising and even faster configurations could be found around such Pareto points. Using greedy algorithm could be very time effective

in limiting the large exploration space and producing better results. However, the problem with greedy algorithms is that they select the first best result ending up in local optima point as found in [15].

Heading towards fully automated exploration tool, the user-defined resource unit amounts and threshold parameters are no longer required, as opposed to our previous exploration implementation [5]. To define the desired resulting architecture the designer now has to input just two limiting real-time execution performance parameters (Pareto points): clock cycle count and operation frequency. Clock cycle count can be used to successfully prune the architectures which reach it and a synthesis tool is run to check if the frequency requirement is satisfied. The value of these parameters can be manually decided by the designer, but can also be found out by sweeping different values to find the optimal starting Pareto points, where the performance of the output configurations no longer increase. The new algorithm explores much wider exploration design space, especially around those Pareto points and it can now produce multiple suitable architectures. As a result, the designer has several choices to select either architectures with minimal hardware resources or slightly larger ones but with extra performance still satisfying the input parameters.

The exploration pipeline is depicted in Fig. 4 and its simplified heuristics are described in the Algorithm 1. It contains multiple resource pruning and optimization passes, which produce lists of different architecture configurations in the order of increasing resources. Configurations which do not fulfill the clock-cycle count requirements specified by the designer are pruned and only the promising ones are considered as candidates and passed forward the pipeline. Only single configuration with minimal resources is passed at a time to the next pass. The architectures are picked starting from the smallest in terms of resources (FUs, RFs, buses, RF ports). When the end of the pipeline is reached, the configuration is tested for conforming to both input parameters and marked as promising. After that, the exploration continues going backwards in the pipeline to the previous pass. If the previous pass has some configurations left in the produced set, the algorithm picks the next with more resources and continues the exploration by passing that configuration to the next pass.

Algorithm 1: Exploration heuristics algorithm

```
while # of found configurations amount limit reached
 do
    if not reached end of exploration pipeline then
        if there are configurations left then
            run pass;
            prune not satisfying configurations;
            pick smallest configuration;
            advance forward in the pipeline;
        else
            prune whole configuration set in previous
              pass;
            jump backwards in the pipeline;
        end
    else
        run synthesis;
        if synthesis results are good then
            add configuration to final results;
        else
            prune whole configuration set in previous
              pass;
            jump backwards in the pipeline;
        end
    end
end
Result: List of found promising design
        configurations.
```

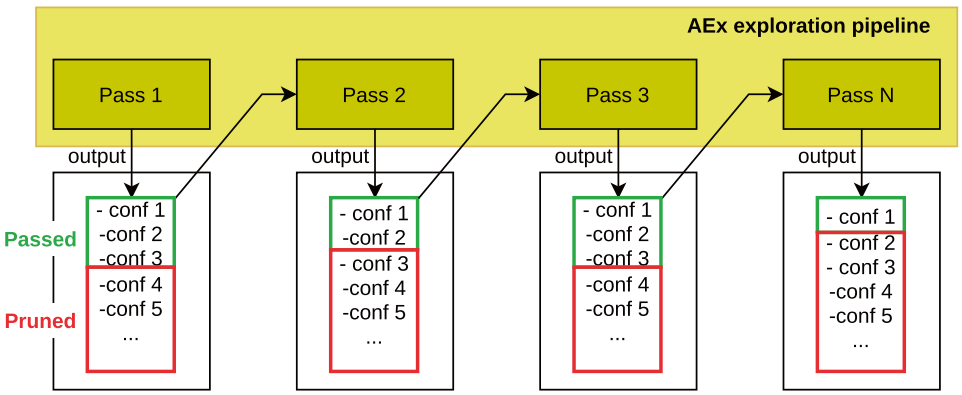


Figure 4 AEx exploration pipeline. Each architecture optimization pass produces multiple design choices aka configuration sets in the growing order of resources. Some of the configurations are pruned based on designer performance restrictions and others are passed to the next optimization pass as an input one by one producing subsets of their own. At the end of the pipeline configurations are synthesized and pruned if the design goals are not met. This might lead to more pruning in the preceding sets, as picking configurations with more resources would also fail.

The exploration pipeline deals with the branching passes by performing a depth-first search to the tree, where the nodes at each pass are ordered by their resource count from smaller to larger. An example run of this algorithm is shown in Fig. 5. During the exploration process the back and forth movement through the exploration tree produces tens or even hundreds of promising configurations. Since each branching pass can create tens of configurations, the full exploration of the tree can take a very long time. To limit this, the designer can set the desired number of promising configurations using an input parameter. After reaching that number of promising configurations, the exploration can stop. Since the configurations are evaluated in the order of growing resources, the chosen configurations at that point are the ones with minimal resources. While evaluating the clock frequency of a configuration, the synthesis might fail because of over-utilization of resources or too long of a critical path. At that point, it can be assumed that we don't have to go backwards to the previous branching pass to pick the architecture with increased resource amount, because increasing the resources would increase the critical path and the operating frequency will also fail with this new subset. Based on our observations, this measure has a positive effect in dropping out more possible design space points and minimizing the overall exploration time.

The described exploration algorithm is not strictly specific to TTAs. However, many of the optimization passes are TTA-specific. Therefore it could be possible to use the same algorithm for different types of instruction set architectures by removing and adding new passes to it.

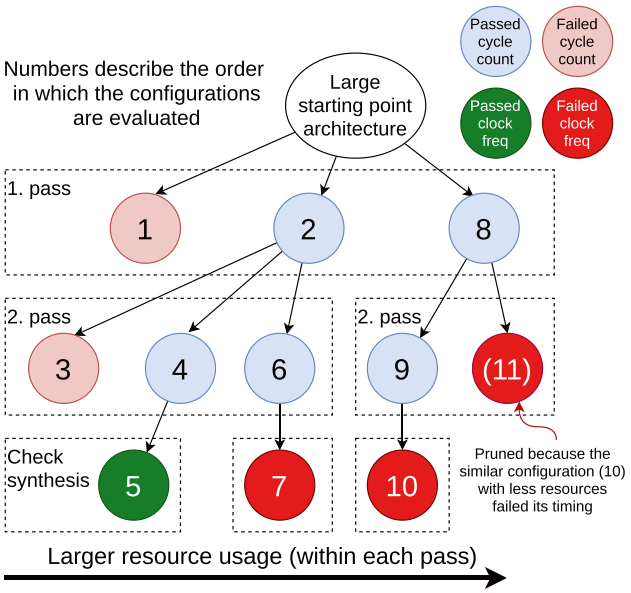


Figure 5 A simplified example of an exploration run with 2 branching passes and clock frequency check using synthesis. At the end, only the configuration number 5 is returned as a promising configuration.

4 Evaluation

The described HLS flow was implemented to a retargetable tool flow called TCE [16]. It has a design space exploration framework which allows defining new automated modification algorithms to perform the iterative search of the design space [17].

4.1 Setup

To benchmark the efficiency of our new exploration heuristics we run the same benchmarks as in our previous work [5]. We used maximum operating frequency and clock-cycle counts from the previous work's performance-oriented architectures as input parameters for the new architecture generation. To compare the results we use execution time, FPGA resource usage in LUTs, maximum operating frequency, instruction width and instruction memory size. The comparison is made against our previous work, MicroBlaze soft-core processor, ARM Cortex-A9 processor and fixed-function accelerators generated by HLS tool Vitis by Xilinx [18].

The exploration was run on CHStone [19] benchmark suite, which is a set of various applications of different domain, such as multimedia, signal processing and cryptography. We generated 8 application specific architectures, one for each benchmark, picking the ones with best execution time performance. The synthesis runs were performed with Vivado using Zynq 7020 FPGA device as a target. Latencies for the most arithmetic operations and constant shift were set to 2 cycles, dynamic shift 3, memory load operations 4 and same for complex operations like multiply and division.

4.2 Exploration Results

The increased exploration design space and additional synthesis runs to check operating frequency resulted in increased total exploration time, but thanks to the successful pruning heuristics we managed to keep them at the reasonable level. Overall average exploration time has increased around 1.5-2 times from our previous work. The number of desired promising architectures significantly affects the exploration time, since the exploration can be stopped once that number of configurations has been found.

Estimating the operating frequency of an architecture description without performing the actual synthesis could improve the exploration time greatly, as synthesis times for each configuration are an order of magnitude slower than the compilation and simulation times. This type of estimator could be created by collecting the timing information of operations and other configuration parameters into a database, and intelligently combining them together. Alternatively, creating a data-driven estimator would be possible,

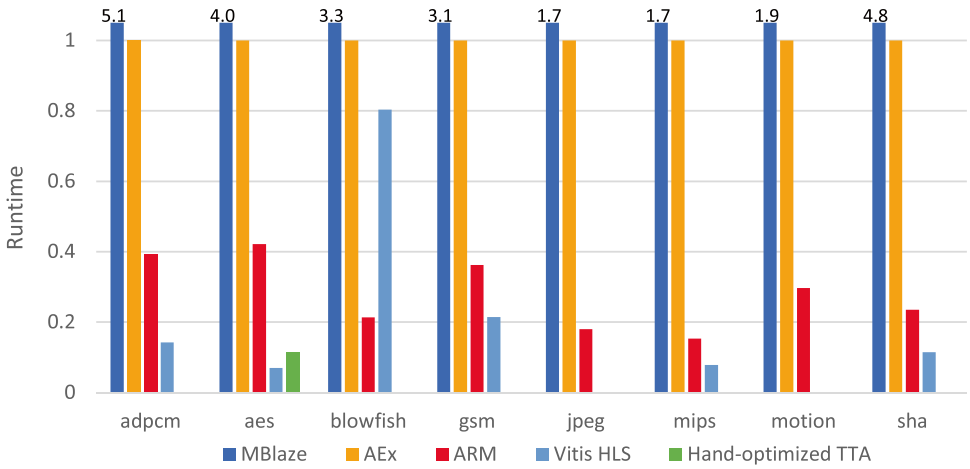


Figure 6 Overall runtime comparison between MicroBlaze, AEx, Arm and Vitis HLS tool. MicroBlaze runtime values are truncated and the actual value is given at the top of the bar. Hand-optimization was only performed for aes benchmark.

but would require thousands of synthesized configurations as training data.

Now, with our improved exploration heuristics we are able to explore a wider design space around given Pareto points. This results in 5 completely new benchmark architectures, while the architectures for adpcm, blowfish and gsm stayed the same compared to our previous work. Table 1 summarizes the synthesis results for the generated architectures. The overall look-up-table (LUT) utilization is improved around 11%. This is because the algorithm picked slightly smaller architecture having 1 function unit, bus or register file port less, while having around the same or better performance. The maximum operating frequencies stayed around the same level with 1-3% improvement. Table 2 presents the instruction widths in the produced architectures. The instruction width has decreased in the most new generated architectures. This is natural, since the core with less resources also has a shorter instruction word. As a result, the total instruction memory usage decreased on average by 16%.

Figure 6 shows overall execution performance of AEx against MicroBlaze soft-core processor, ARM Cortex-A9 processor and Vitis. MicroBlaze performance is worse by a magnitude of 2 to 5, but it's understandable as it is a simple scalar processor and not oriented for performance. The discussion on Vitis results is continued in the next section.

The dual-issue superscalar ARM Cortex-A9 [20] is chosen as one of the comparisons because it's available as a hard processor on the used Zynq 7020 SoC. Additionally, as the proposed method is a processor-based approach, it makes sense to evaluate the differences to a hard processor core. In these benchmarks, ARM outperforms AEx by 60% to 80%, which is mainly explained by the higher clock frequency of 650 MHz [21]. The difference in the clock frequencies seems to match quite closely with the difference in performance. Since the limitations of scaling the performance of a multi-issue processor by increasing the available ILP are well known [22, 23], there simply aren't that many available operations that can be scheduled in parallel, thus ILP is very limited. Compared to the ARM processor with much higher clock frequency, the proposed method should sustain 3x more operations in parallel than ARM. This leads to an impossible situation, where there simply isn't any more available ILP in the program to exploit.

Total instruction memory size relative to AEx is depicted in Fig. 7. AEx has lower instruction memory usage than MicroBlaze and ARM in blowfish and sha benchmarks. Reason for this is the significantly lower number of instructions, even though the instruction word is much wider. Thanks to the application specific tailoring of the architectures, the well-known TTA's problem of inflated instruction memory size is kept here at reasonable levels.

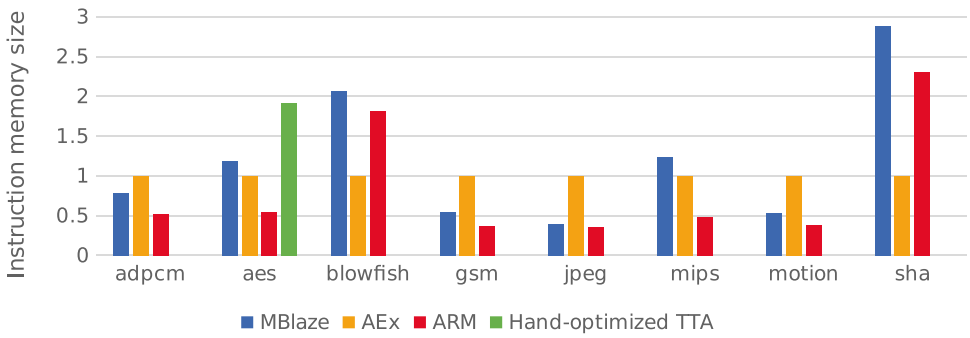


Figure 7 A comparison of instruction memory sizes. Hand-optimization was only performed for aes benchmark.

Table 1 Synthesis results: LUT utilization, maximum operating frequency and energy estimation. Comparison is made with our previous work against updated AEx. The LUT utilization is further broken down to the LUTs used by the processor interconnection (IC) and register files (RF).

Benchmark	Core	IC	RF	FMax (MHz)	Energy (μJ)
adpcm	1756	562	89	198	8.57
aes	**2226 (1.01x)**	**632 (0.86x)**	312	186	3.62
blowfish	1625	469	49	202	74.9
gsm	2565	630	500	180	1.80
jpeg	**1551 (0.75x)**	**392 (0.70x)**	49	205	307
mips	**1269 (0.85x)**	**276 (0.60x)**	25	199	4.57
motion	**1359 (0.92x)**	**390 (0.85x)**	49	192	0.745
sha	**1884 (0.89x)**	**649 (1.05x)**	49	188	49.8

Instruction compression feature of the tool set was not utilized in this research. It could be applied as a post-processing step for the final configurations to bring the instruction memory usage even lower. In the future it could be interesting to couple the instruction compression to be a part of the exploration algorithm, so that the effect of it would be reflected in the exploration results.

4.3 SoC Implementation Assessment

In order to assess the potential for automatically generated co-processors as ASIPs for new SoC designs, we synthesized one of the produced configurations using a modern ASIC technology. In order to put the numbers to a perspective, we compared the proposed method to a well-known general-purpose RISC-V [24] ISA implementation called zero-riscy [25]. The designs are synthesized with Synopsys Design Compiler [26] using a 28 nm process. Zero-riscy is a small energy optimized core, thus its numbers should be taken as a coarse grained design point in terms of small consumption general purpose core. The results of the ASIC synthesis are presented in Table 3.

Table 3 Comparison of the machine AEx generated for aes benchmark. A small general purpose RISC-V implementation Zero-riscy given as a reference point.

	Zero-riscy	AEx-generated core
FMax (GHz)	1.33	3.33 **(2.5x)**
Peak arithmetic ops (Gops/s)	1.33	6.67 **(5.02x)**
Cycle count (aes)	31 873	23 246 **(0.73x)**
Estimated runtime (ns)	23 904	6 973 **(0.29x)**
Area (μm^2)	11 731	17 865 **(1.52x)**

The core generated with the proposed method is able to reach 3.3 GHz clock frequency while not demanding significantly larger area than the comparison target. This is because the specialization of the processor architecture allows it to have a simpler hardware than the general purpose core. Since the core generated by the proposed method is able to issue multiple operation moves in parallel, it's able to launch 2 arithmetic operations in one cycle. This gives it an estimated 5x advantage in peak arithmetic ops compared to the single-issue zero-riscy.

In the aes benchmark, the proposed specialized co-processor is able to reach over three times higher performance, while still remaining software programmable. It is important to note that the proposed method uses only a set of quite standard basic operations, which means that it remains highly software programmable for other similar applications.

Table 2 Instruction widths and their relative sizes in comparison to previous work.

	Instruction width (bits)	
benchmark	**Previous work**	**Current work**
adpcm	80	80
aes	115	**91 (0.79x)**
blowfish	63	63
gsm	101	101
jpeg	83	**64 (0.77x)**
mips	59	**44 (0.75x)**
motion	59	**58 (0.98x)**
sha	82	**83 (1.01x)**

Table 4 Runtime, utilization and power results from Vitis HLS compared to the AEx results. Vitis HLS isn't able to synthesize jpeg and motion benchmarks.

Kernel	Fmax	Runtime (ns)	LUT	Energy (μJ)
adpcm	131	47107 (0.14x)	12378 (7.0x)	11.4 (1.3x)
aes	235	8809 (0.070x)	955 (0.43x)	0.132 (0.036x)
blowfish	145	3009503 (0.80x)	2591 (1.6x)	126 (1.7x)
gsm	140	13786 (0.22x)	4723 (1.8x)	1.67 (0.93x)
jpeg	-	-	-	-
mips	159	19975 (0.079x)	1634 (1.3x)	0.539 (0.12x)
motion	-	-	-	-
sha	168	260821 (0.12x)	13372 (7.1x)	121 (2.4x)

5 Comparison to Vitis HLS

In order to estimate the quality of the results, a comparison to a state of the art HLS-method should be included. The CHStone benchmarks are synthesized using Xilinx Vitis HLS tool [18] (version 2020.2) targeting the Zynq 7020 FPGA device. The benchmarks are synthesized using the default C synthesis settings with no significant source code modifications. The results are presented in Table 4. Even though the adpcm benchmark is valid C code, it produces the wrong result in Vitis's C/RTL-cosimulation. This is probably because it's using some feature of C unsupported by Vitis HLS. The numbers for it are presented anyway for comparison.

It's important to note that since the CHStone benchmarks contain both the input data and the output checking functionality as part of their source code, there are no external data interfaces coming out of the kernels. If the kernels were to be used with dynamic input data, the data interfaces would need to be explicitly defined by the user. Another limitation of the HLS tool is the missing support for pointer-to-pointer structures, which causes two of the benchmarks (jpeg and motion) to be non-synthesizable without source code modifications. In the proposed method, the external interfaces are easily understandable regular memory interfaces, and the pointer-to-pointer-structures are supported naturally as part of the C language.

However, on the other benchmarks that it manages to synthesize, the Vitis HLS is able to reach significantly better runtime performance than the proposed runtime programmable co-processor-approach. Some amount of performance overhead is to be expected, since the proposed approach is a complete software programmable processor. The LUT utilization seems to vary between the benchmarks more than it did for the generated TTA processor. This is probably due to the different amounts of automatic unrolling the HLS tool performs, duplicating hardware to execute multiple loop iterations in parallel.

Energy estimates seem to correlate naturally to the runtime and the area utilization values. Since the Vitis HLS generated designs can finish the application faster, they use a smaller total amount of energy. On the other hand, if the LUT utilization is larger, it increases the power usage. On three of the benchmarks (adpcm, blowfish and sha), this causes the energy usage be larger than with the proposed AEx-generated cores, even though they execute the application faster. At the other end of the spectrum, the aes benchmark executes significantly faster while using less area and power which leads to it having just 3.6% energy usage of the AEx generated core.

Comparing against the Vitis HLS-approach can give insight on how to improve the proposed method further. The relatively worst performing benchmark aes is chosen for further analysis (around 14.2x slower than Vitis). In the following subsections we analyze a few of the features that have the potential to significantly increase the performance of the co-processor-approach. This manual optimization shows that the TTA-based co-processor approach is able to almost reach the performance of a commercial HLS tool. The results of the manual optimization are summarized in Table 5. None of the implemented manual optimizations would be impossible to perform automatically within AEx. Therefore, this level of performance increase can be estimated to be a realistic target for further AEx improvements.

Table 5 The improvement in the runtime of aes benchmark with manual optimizations.

Cumulative optimizations	Runtime (ns)	Compared to Vitis runtime
AEx	124 978	14.19
Unroll	78 930	8.96
Custom operations	19 179	2.18
Larger machine	14 454	1.64

5.1 Loop Pipelining

Loops are a very common structure in data processing programs. For example, in Vitis HLS tool, a lot of attention is given to the loop initiation intervals and lengths of loops, as the effects of those get multiplied by the iteration count of the loop. Exploiting the parallelism between independent iterations of a loop is also one way to gain more instruction-level parallelism (ILP) to the program. Since TTA is an exposed datapath architecture where the ILP must be utilized at compile-time, the compiler must perform the interleaving of the loop iterations using e.g. software pipelining [27]. Improvements in the compiler will automatically propagate to the exploration of better architectures.

However, as the TCE compiler's current ability to perform software pipelining is still rather limited, aggressive loop unrolling can be used as an alternative way to exploit loop-level parallelism at the cost of instruction memory size. After applying more aggressive loop unrolling, the runtime was brought down to only be 8.96x slower than the Vitis runtime. Naturally, this comes at the cost of increased instruction memory usage, which is brought up by 39%.

5.2 Automated Generation of Custom Operations

Since aes is a very bit manipulation-oriented algorithm, it would benefit from special operations that perform multiple basic operation chains within a single cycle. This increases the amount of operations executed per cycle by pushing more of the operations inside a single cycle at critical points of the

program. At the moment, the proposed exploration method is only able to automatically use a set of commonly used operations that roughly match C language's operators (add, sub, shifts, xor, etc.). Thus, even simple constant shifts and chains of subword logic operations that could fit inside a single clock cycle now consume at least a single clock cycle each.

In order to assess the performance potential of custom operations, two custom operations were created manually based on the insides of the most critical loops. The custom operations can be described as directed acyclic graphs (DAG) recursively consisting of simpler sub-operations [28]. The hardware for these operations can then be generated automatically based on the DAG. After adding the custom operations to the machine generated by AEx and calling them manually from the source code, the runtime of aes benchmark is brought down to only be 2.18x slower than the Vitis runtime. As a final optimization step, a slightly larger machine is manually created to better take advantage of the ILP exposed up by the loop unrolling, the runtime is further brought down to only be 1.64x slower than Vitis.

Currently, the custom operations need to be defined by the designer manually. This optimization example demonstrates the performance gains available from custom operations. It shows that adding support for automatic discovery of the custom operations to the exploration pipeline seems to be an important next step. This is required to make the method scalable to new unknown applications, without requiring the designer input in defining the custom operations.

At the time of writing, the source code of the program has to be manually edited to use the custom operations due to the ineffectiveness of the instruction selector in choosing more complex operations [28]. In the future, the compiler should be improved to automatically utilize these custom operations. This is required so that the custom operations created to improve the performance of a single application would be automatically usable in other applications. Specialization level of the custom operations is an important parameter to keep in mind when tailoring for certain application while remaining software programmable for other similar applications.

5.3 Parallel Memories

So far in the proposed approach, only a single load-store unit is used to access the data memory, which resides in a single on-chip block RAM. This can easily create congestion as there are often a lot of memory operations at the critical parts of the program that could be performed in parallel. Memory components often support multiple parallel ports, so utilizing all of them seems obvious to maximize the parallel data transfers to memory. Adding a second LSU to the architecture that connects to the second port of the single data memory block RAM would be the easiest way to accomplish this.

In Vitis HLS tool, different variables in the input code can be allocated to separate memory components. Even further, array partitioning is a well-known optimization in HLS to split a large array into multiple memory components to get parallel access to the array. This is combined with the memory access pattern analysis to ensure that there are no conflicting accesses to the same memory component. This can help e.g. in loop pipelining to help parallelize the loop iterations by removing the conflicting memory accesses.

Adding multiple separate memory components to the proposed method would require a lot of automatic analysis to split the variables between the memory components. Additionally, interfacing with the outside world becomes more difficult the more fragmented the memory system is. One of the advantages of a software programmable accelerator is the ability to support multiple different applications. Therefore, the memory hierarchy tailoring should be done carefully to not overfit the memory hierarchy to certain application's access pattern and lose the benefits of a software programmable processor. However, it's possible that there would be some generic access patterns that the processor could still be tailored to (e.g. stencil).

Alternatively, the compiler already supports OpenCL C, so changing the input language to OpenCL could provide more tools to the handling of different memory components. In OpenCL, the buffers could be allocated to different memory components at runtime.

5.4 Data-level Parallelism

As described above, there often isn't enough ILP to exploit in common C programs, including the CHStone benchmarks. To get around this, more data-level parallel programs can be used. Therefore, a change to more parallel input language such as OpenCL might be necessary to fully realize the benefits of customizable architectures. Manually designed TTA-cores have already been shown to perform better than the ARM Cortex-A9 in certain OpenCL benchmarks [29], so adding OpenCL support to AEx seems like natural progression.

One simple way to support the data-level parallelism of OpenCL would be to support the vector datatypes of OpenCL using SIMD operations. These SIMD operations could be highly specialized for the given application to reduce the resource requirements coming from the wide datapaths. There would only be minimal changes needed to the exploration pipeline to accommodate the automatic exploration of architectures with both SIMD and scalar operations.

5.5 Discussion

Based on these findings, the lowest hanging fruits to get on par with the commercial state of the art is to automate the discovery and use of custom operations as well as improve the software pipelining support of the compiler. Another

alternative would be to add a more intelligent heuristics for controlling the software pipelining/loop unrolling aggressiveness based on the available instruction memory.

Based on our analysis of an example benchmark, these improvements would likely heavily improve most, if not all of the other benchmarks as well, bringing the performance close to the fixed function implementations generated by the Vitis HLS tool without suffering from its limitations placed on the supported programming language constructs.

6 Related Work

In a similar work, the authors apply same shrinking technique to the large VLIW architecture and explore architectures based on the number of issue-slots, number of parallel FUs, RF and memory sizes [14]. Underutilized units are merged into other and oversized register files are resized to minimize the instruction width. To prune the large exploration space the additional genetic algorithm framework is applied. This brings several obstacles, as it requires additional configuration settings to be applied for achieving optimal results. Also authors experienced same prolonged evaluation time difficulties for larger algorithms and applied result caching for it.

In another work the authors experience the same issues with the large design space using CGRA architectures where they apply statistical approach to solve the operating frequency estimation issues [30]. They created a large database of different CGRA architectures and generated Verilog code for it. A part of initial designs might be inefficient and are pruned using LUT estimator. After that, they synthesize the remaining results in the database, which took several months to complete on the multiple Vivado instances. Using the results they created the statistical estimator for architecture parameters affecting the clock frequency with an accuracy between 2-5%.

There has been a lot of activity in the HLS research field lately, involving both academic and commercial HLS tools. Most tools fall into two language categories, general purpose programming languages (GPL) such as C or C++ or target specific languages like SystemC, which are derived from GPLs with specific structures to describe hardware information to the tools. Inserting additional hardware hints to the tools usually squeezes more performance improvement, but requires additional hardware knowledge from the developer and has multi-platform portability issues. A more sensible way is to generate hardware descriptions from the code written in high-level language without any original application code modifications using HLS tools. The code is handled using intermediate representation (IR) and optimized using several common software code optimizations before final hardware RTL description generation for specific target platform.

Commercial tools like Xilinx's Vitis HLS and Catapult C by Mentor Graphics offer hardware code generation from the HLL applications written in languages like C, C++, Python or SystemC. Where Catapult empowers developers with full control over HLS synthesis, generating bug free and power efficient RTL with additional verifications, Vitis provides even richer ecosystem. Vitis compiler is based on the LLVM [31], which is an open source compiler toolchain providing powerful code optimizations via multiple HLL code transformation passes. Open source application acceleration libraries offer out-of-the-box acceleration with minimal to zero-code changes to the existing applications. To leverage the learning curve of feature rich tools the vendors provide good support and online tutorials.

Academic tools LegUp, Bambu and DWARV deliver powerful functionality, but are lacking commercial tools features and their target platforms support is limited [32–34]. LegUp, once an open-source HLS tool, but now acquired by Microchip, translates application C, C++ code into intermediate representation for easier handling to perform multiple compiler optimizations. One of its powerful features is pthread, OpenMP parallel code support and it can generate multiple hardware cores without any annotations to the original code, so the application can be run concurrently. Bambu utilizes GCC compiler to perform code optimisations and can support several target platforms. DWARV is based on CoSy compiler and can perform over two hundred different compiler optimizations.

A number of surveys show that the performance gap between the commercial and academic HLS tools is not that significant and the differences fall mostly in usability and feature availability [3, 4]. Commercial tools usually provide more pleasant usability, optimization features and support several platforms or languages. Results also indicate that there is still no single ultimate HLS tool that would perform equally well in every possible application domain. Developers still need to know hardware specific tricks and describe them to the HLS tools, to realize them efficiently in the target platforms, by either modifying the original application code or specifying tool specific parameters.

7 Conclusions

In this paper we introduced AEx, an automated high-level synthesis tool for programmable co-processors. We generated application specific architectures using new exploration heuristics and compared the results to our previous work, MicroBlaze soft-core processor, ARM processor and Vitis HLS tool. We synthesized one of the architectures on ASIC technology to demonstrate the potential for SoC use case by showing 71% runtime reduction with 52% area increase. New heuristics operate on a wider design space, but still stay in reasonable time limits, generating sensible architectures in terms of performance. Comparison to modern HLS tool highlighted the key spots to improve in the AEx tool, which we will focus on in the future work. With added manual fine-tuning

the performance slowdown was brought down to only 1.64x compared to the fixed-function accelerator generated by the HLS tool.

In the future, the tool run time can be improved with earlier resource and performance estimation which reduces the amount of synthesis runs required. Additionally, feeding back the resource constraint information from the compiler to the exploration pipeline could help to speed up the search times and to find even more optimal architectures. Discovery and automated generation of custom operations could significantly improve the performance of the generated configurations. Moving to more data parallel programs defined using the OpenCL standard could help to improve performance by exposing more of the parallelism to the exploration algorithm and the compiler.

Acknowledgements The work for this publication was funded by ECSEL Joint Undertaking (JU) under grant agreement No 783162 (FitOptiVis [6]). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Netherlands, Czech Republic, Finland, Spain, Italy. It was also supported by European Union's Horizon 2020 research and innovation programme under Grant Agreement No 871738 (CPSoSaware) and Academy of Finland (decision #331344).

Funding Open access funding provided by Tampere University including Tampere University Hospital, Tampere University of Applied Sciences (TUNI).

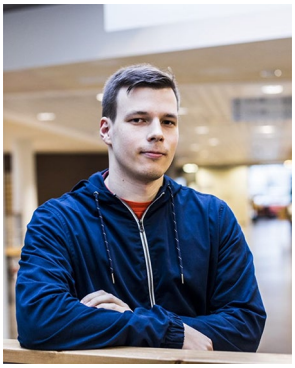
References

1. Fahmy, S. A., Vipin, K., & Shreejith, S. (2015). Virtualized FPGA accelerators for efficient cloud computing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 430–435.
2. Ren, H. (2014) A brief introduction on contemporary high-level synthesis. In *Proceedings of the 2014 IEEE International Conference on IC Design & Technology (ICICDT)*.
3. Lahti, S., Sjövall, P., Vanne, J., & Hämäläinen, T. D. (2018). *Are we there yet?* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems: A study on the state of high-level synthesis.
4. Nane, R., et al. (2016). A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *35*(10).
5. Hirvonen, A., Tervo, K., Kultala, H., & Jääskeläinen, P. (2019). AEx: Automated customization of exposed datapath soft-cores. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pp. 35–42.
6. Al-Ars, Z., et al. (2019). The FitOptiVis ECSEL project: Highly efficient distributed embedded image/video processing in cyber-physical systems. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19, pp. 333–338, New York, NY, USA. Association for Computing Machinery.
7. Hoogerbrugge, J., & Corporaal, H. (1994). Register file port requirements of Transport Triggered Architectures. In *IProceedings of the 27th Annual International Symposium on Microarchitecture*.
8. Corporaal, H., & Hoogerbrugge, J. (2002). *Code Generation for Transport Triggered Architectures*, pp. 240–259. Springer US, Boston, MA.
9. Jääskeläinen, P., Kultala, H., Viitanen, T., & Takala, J. (2014). Code density and energy efficiency of exposed datapath architectures. *Journal of Signal Processing Systems, 80*(1), 49–64.
10. Jääskeläinen, P., Tervo, A., Vayá, G. P., Viitanen, T., Behmann, N., Takala, J., & Blume, H. (2018). Transport-triggered soft cores. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*.
11. Palesi, M., & Givargis, T. (2002). Multi-objective design space exploration using genetic algorithms. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, pp. 67–72, New York, NY, USA, 2002. Association for Computing Machinery.
12. Kumar, A., & Chakarverty, S. (2011). Design optimization using genetic algorithm and cuckoo search. In *2011 IEEE International Conference on Electro/Information Technology*, pp. 1–5.
13. Ferrandi, F., Lanzi, P. L., Loiacono, D., Pilato, C., & Sciuto, D. (2008). A multi-objective genetic algorithm for design space exploration in high-level synthesis. In *2008 IEEE Computer Society Annual Symposium on VLSI*, pp. 417–422.
14. Jordans, R., Jóźwiak, L., & Corporaal, H. (2014). Instruction-set architecture exploration of VLIW ASIPs using a genetic algorithm. In *2014 3rd Mediterranean Conference on Embedded Computing (MECO)*, pp. 32–35.
15. Viitanen, T., Kultala, H., Jääskeläinen, P., & Takala, J. (2014). Heuristics for greedy transport triggered architecture interconnect exploration. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. ACM.
16. Jääskeläinen, P., Viitanen, T., Takala, J., & Berg, H. (2017). HW/SW co-design toolset for customization of exposed datapath processors. *Computing Platforms for Software-Defined Radio*, 147–164.
17. Esko, O., Jääskeläinen, P., Huerta, P., dela Lama, C. S., Takala, J., & Martinez, J. I. (2010). Customized exposed datapath soft-core design flow with compiler support. In *Proceedings of the International Conference on Field Programmable Logic and Applications*.
18. Xilinx. *Vitis High-Level Synthesis User Guide (UG1399)*. Retrieved February 14, 2023, from https://docs.xilinx.com/r/en-US/ug1399-vitis-hls
19. Hara, Y., Tomiyama, H., Honda, S., & Takada, H. (2009). Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing, 17*, 242–254.
20. ARM. *Cortex-A9*. Retrieved February 14, 2023, from https://developer.arm.com/ip-products/processors/cortex-a/cortex-a9
21. Digilent. (2017). *PYNQ-Z1 Board Reference Manual*.
22. Jouppi, N., & Wall, D. (1989). Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the third international conference on architectural support for programming languages and operating systems*, ASPLOS III, pp. 272–282. ACM.
23. Wall, D. W. (1991). Limits of instruction-level parallelism. *SIGPLAN Notices*, *26*(4), 176–188.
24. Waterman, A., Lee, Y., Patterson, D. A., & Asanovic, K. (2011). *The RISC-V instruction set manual, volume I: Base user-level ISA*. EECS Department, UC Berkeley, Technical Report UCB/EECS-2011-62.

25. Schiavone, P. D., Conti, F., Rossi, D., Gautschi, M., Pullini, A., Flamand, E., & Benini, L. (2017). Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for internet-of-things applications. In *Proceedings of International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*.
26. Synopsys. *Design Compiler Graphical*. Retrieved February 14, 2023, from https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html
27. Lam, M. (1988). Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 conference on programming language design and implementation*, PLDI '88, pp. 318–328. ACM.
28. Kultala, H., Jaaskelainen, P., & Takala, J. (2011). Operation set customization in retargetable compilers. In *2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pp. 761–765. IEEE.
29. Tervo, K., Malik, S., Leppanen, T., & Jääskeläinen, P. (2020). TTA-SIMD soft core processors. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 79–84.
30. Wolf, D. L., Spang, C., & Hochberger, C. (2020). Towards purposeful design space exploration of heterogeneous CGRAs: Clock frequency estimation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6.
31. Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 20–24. Palo Alto, CA.
32. Nane, R., Sima, V., Olivier, B., Meeuws, R., Yankova, Y., & Bertels, K. (2012). DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*.
33. Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., & Czajkowski, T. (2011). LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, New York, NY, USA. ACM.
34. Pilato, C., & Ferrandi, F. (2013). Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *23rd International Conference on Field Programmable Logic and Applications*.

Alex Hirvonen received the MSc degree in Electrical Engineering from Tampere University, Tampere, Finland in 2021. He works currently as a Software Engineer at M-Files, Finland on topics related to efficient data storage management.

Topi Leppänen received the MSc degree in Electrical Engineering in 2021 from Tampere University, Finland, where he is currently pursuing a PhD degree. His research interests include heterogeneous platforms and hardware acceleration. His current research focuses on easier development and programming of diverse systems with specialized programmable and nonprogrammable accelerators.



Kari Hepola received the MSc degree in Electrical Engineering from Tampere University, Tampere, Finland in 2022, where he is currently working towards the PhD degree. His current research interests include multiple instruction set architectures and application-specific instruction set processors with the goal of increasing energy efficiency and flexibility of programmable processors.

Joonas Multanen received his MSc degree in Electrical Engineering in 2015 from Tampere University of Technology and his PhD degree in 2021 from Tampere University (TAU), Finland. He is currently a postdoctoral researcher at the Faculty of Information Technology and Communication Sciences in TAU. His research interests include energy efficient computer architectures.

Joost Hoozemans received his BSc in Computer Science from Utrecht University in 2011 and his MSc and PhD in Computer Engineering from Delft University of Technology in 2014 and 2018, respectively. His research interests include VLIW and TTA processors, reconfigurable computing, FPGA programmability and dataflow computing.

Pekka Jääskeläinen (Associate Professor) leads the Customized Parallel Computing group of Tampere University. He has worked on heterogeneous platform customization and programming topics since the early 2000s. In addition to his academic publication activities, he is responsible for two heterogeneous computing related open-source projects; OpenASIP and Portable Computing Language (PoCL). His current research interests include methods and tools to reduce the engineering effort involved in design and programming of diverse heterogeneous platforms and hardware, and compiler techniques to reduce the energy consumption of programmable processors.