AEx: Automated Customization of Exposed Datapath Soft-Cores

Alex Hirvonen, Kati Tervo, Heikki Kultala, Pekka Jääskeläinen
Tampere University
Tampere, Finland
email: {alex.hirvonen,aleksi.tervo,heikki.kultala,pekka.jaaskelainen}@tuni.fi

***Abstract*—High-level synthesis tools aim to produce hardware designs out of software descriptions with a goal to lower the bar in FPGA usage for software engineers. Despite their recent progress, however, HLS tools still require FPGA target specific pragmas and other modifications to the originally processor-targeting source code descriptions.**

Customized soft core based overlay architectures provide a software programmable layer on top of the FPGA fabric. The benefit of this approach is that a platform independent compiler target is presented to the programs, which lowers the porting burden, and online repurposing the same configuration is natural by just switching the executed program. The main drawback, like with any overlay architecture, are the additional implementation overheads the overlay imposes to the resource consumption and the maximum operating frequency.

In this paper we show how by utilizing the efficient structure of Transport-Triggered Architectures (TTA), soft-cores can be customized automatically to benefit from the flexible FPGA fabric while still presenting a comfortable software layer to the users. The results compared to previously published non-specialized TTA soft cores indicate equal or better execution times, while the program image size is reduced by up to 49%, and overall resource utilization improved from 10% to 60%.

***Index Terms*—FPGA, soft cores, programmable overlays, high level synthesis, Transport-Triggered Architecture**

I. Introduction

With multiple cloud providers now offering FPGA acceleration among their options, accessibility of FPGAs is on the rise. While the threshold for getting access to reconfigurable logic hardware is getting lower, their design and programming tools lag behind and should still evolve to comfortably accommodate a wider audience of developers.

Traditionally, *register transfer level (RTL)* descriptions have been used to describe very large scale integration systems. Digital hardware description languages such as VHDL and Verilog allow designer to describe such systems in terms of registers and low-level logic and arithmetic functions with a low level of abstraction. Designing, evaluating, and especially verifying, such systems usually takes a significant amount of time. In contrast, implementing algorithms in a higher level software programming languages provides a shorter learning curve and increased engineering efficiency thanks to their higher abstraction level [1], [2].

High level synthesis tools provide the ability to generate RTL descriptions from applications written using a *high level language (HLL)*. Raising the level of abstraction gives software engineers potential to design VLSI systems with or without digital hardware design expertise. Cutting down development and evaluation time allows faster prototyping and shorter time-to-market [3], and in case of FPGA chips, lowers the bar of their adoption to software developers without hardware design backgrounds. Despite their decent success, however, HLS tools still require FPGA target-specific pragmas and other modifications to the originally processor-targeting source code descriptions. This reduces portability of the implementation and adversely affects the learning threshold due to the need for the developers to learn the specific description differences of each targeted FPGA platform.

Customized soft core based overlay architectures provide a software programmable layer on top of the FPGA fabric. The benefit of using a soft core layer is that a familiar feeling instruction-set architecture target is presented to the programmer, which is expected to lower the portability burden. In addition, repurposing the same configuration is natural by just switching the executed program, thus multi-function designs are supported via software reprogramming. The main drawback, like with any overlay architecture, are the additional implementation overheads the overlay imposes to the resource consumption and maximum operating frequency. In the software programmable overlays, one of the key overheads is the program image that consumes additional memory typically in an on-chip storage for fast access.

Previously, it has been shown that *transport-triggered architectures (TTA)* can be implemented efficiently as soft cores thanks to their simplified *register files (RF)* and fine-grained compiler-based control hardware [4]. In this paper we show that by utilizing the efficient and modular structure of TTAs, customized soft-cores can be generated *fully automatically* to benefit from the flexible FPGA fabric while still presenting a comfortable software programmable end result to the developers. We refer to the presented automated exploration techniques as **AEx** later in this paper.

The paper is organized as follows. Section II visits the closest references to the presented work. Section III details TTA, the "processor template" used as the basis in this work. Section IV describes the architecture exploration steps in the proposed soft core generation flow. Section V shows how the final RTL is produced from the end results of the architecture generation. Section VI quantifies the end results of the automated custom processor generation flow. The paper is concluded and future work planned in Section VII.

II. Related Work

There are various academic and commercial HLS tools available. Such tools can be divided into ones that either input target-specific languages or general purpose programming languages, such as C or OpenCL C. Implementing an application in a target-specific language typically improves the results, but requires additional target knowledge from a software engineer and is prone to errors. In practice, this the target adaptation is done by inserting target specific hints to the input of the HLS tools. A more generic approach is to take an application source code written in some high-level programming language and feed it to the HLS tool. The programming language input is the optimized with traditional software optimizations after which they are converted to structures efficient on a particular FPGA.

Commercial tools such as Catapult-C and Vivado HLS offers the ability for designers to work either with C/C++ or SystemC. Both claim to produce efficient RTL code with reduced design-time effort. Vivado's compiler is based on LLVM [5] allowing the designer to benefit from its several code optimizations. Other noticeable features include support for all application domains, floating-point support and automated testbench generation. Different parameters for memory mapping and setting design frequency are also supported.

Academic tools, DWARV, LegUp and Bambu offer powerful functionality to generate efficient RTL code, but are limited in features and supported FPGA devices. [6]–[8]. LegUp translates C code into LLVM IR and performs several compiler optimizations to optimize the generated instructions and targets soft core processors. It also supports pthreads and OpenMP to generate parallel hardware using additional source code annotations. Dwarf is a modular compiler targeting reconfigurable architectures based on the CoSy commercial compiler. It can perform over two hundred different transformation and optimization passes on the IR code generated from C source code. Bambu relies on GCC to perform code optimizations. It can be easily reconfigured to target different devices and technologies. All above academic HLS tools support all application domains and are able to generate testbenches.

Several surveys show that the performance gap between commercial and academic tools is not substantial [1], [3]. The main difference is that the commercial tools typically provide more robust and wider feature sets than the academic tools. For example, they might support multiple input languages and target platforms. According to the surveys, there is no single tool that clearly outperforms in all benchmarks. Thus, to achieve the maximum performance, the software developer still has to understand the specialities in the underlying FPGA chip, and, e.g., know the crucial optimization passes and how to invoke them from the source code or the toolset options.

Our proposed flow is based on a compiler programmable customizable architecture. What makes it unique to the previously proposed ones is its use of TTA as the processor template. The previous work has shown that TTA can produce efficient programmable designs with simpler hardware [9], which has been shown to be the case also with FPGA implementation [4] with manually created designs. In this publication we show that TTA soft core customization can be fully automated when starting from a high-level programming language input.

III. Transport-Triggered Architectures

TTAs are so called *exposed datapath architectures* [10]. The structure of the interconnection network is visible to the programmer through one or more *moves* contained in the instruction word, executed in parallel. Fig. 1 presents the programmer's view of an example TTA processor with five transport buses which allow a maximum of five simultaneous data transports per instruction.

The moves can be constrained to an arbitrary set of input and output ports, each belonging to a *function unit (FU)* or a *register file (RF)*. A move to or from a port belonging to a RF is associated with a register index and implies a read or write at that register. Likewise, a move to a designated *trigger* port in an FU is associated with an opcode, and signals the start of the execution of the corresponding operation.

The move structure is more flexible than the typical operation-triggered format, where all operands are specified in terms of the registers they reside in. In particular, values do not have to be written and read to the RF if they can be moved directly from the FU result port to the operand port [9], [11]. This is further helped by registers in FU ports for storing operands and results, giving the compiler additional freedom to schedule operations.

The exposed datapath of TTAs enables scalable static *instruction level parallelism (ILP)* since multiple FUs can be supported with reduced simpler RFs with reduced port counts. The control logic is further simplified thanks to the programmer controlled register file bypass network; in traditional "register operand based" architectures, the processor needs to automatically determine when to activate a forwarding path for latency saving, while with TTAs it is a programmer responsibility. The self-clear drawback of all the additional programmer control are the additional instruction control bits which bloat the program image.

The connectivity of the *interconnection (IC)* network between FUs and RFs is crucial to the implementation efficiency of TTA designs. Too many connections add instruction bits and also typically results in a longer critical path due to the routing difficulties. A good optimization strategy is to start the optimization process from a connectivity found inside typical VLIW architectures and prune the network as much as the desired generality of the end result allows.

Carefully designed instruction stream constant support is essential in generating high performance TTA designs without wasting instruction bits. In TTAs, buses can have an "immediate field" as one of the source options of the move slot. Having immediates available to multiple buses independently can increase the degree of available ILP, but should be limited in their bit width so as not to increase the instruction width unnecessarily. In practical architectures, the slots are usually

limited to a few bits of width. Due to this, we'll denote them as *short immediates*. An alternative is to define a mode to the instruction encoding that occupies entire move slots to encode a wide constant. This value is loaded by the instruction decoder into an *immediate unit* that acts like an RF with only read ports visible to the programmer controlled interconnection network. The pitfall here is that the move slots for these *long immediates* cannot be used for ordinary moves when loading an immediate value, thus the used move slots should be selected carefully so that common execution patterns aren't disrupted by the immediate.

The customizability of the interconnect along with FU operation sets and RF organization means that the design space of even a streamlined architecture like TTA is quite large. However, the components are modular and each of them are simple to construct on their own. Unlike a typical RISC processor with all or most operations sharing a single pipeline, TTAs isolate portions of the operation set into function units with independent pipelines, like with VLIW architectures. This also allows parallel execution of operations without complex decoding and resource arbitration logic.

IV. Automated Soft Core Architecture Exploration

The proposed automated soft core design space exploration process is initiated with an input application given as C, OpenCL C or LLVM [5] intermediate representation, and a set of design space limiting parameters: 1) The degree of retained instruction-level parallelism (number of parallel arithmetic-logic units (ALU) in the end result) and 2) the maximum clock cycle count allowed. Other optional parameters that shape the end result towards better supporting also other input programs than the one given to drive exploration are also supported. These include immediate size, RF splitting and keeping certain wanted FUs separate.

The design space exploration process outputs an *architecture description file* which describes the processor in adequate detail for the compiler and the simulator. This file can be also used for generating the final RTL implementation, a process which is described in the next section.

The proposed automated design space exploration algorithm for TTA based soft cores is organized as multiple architecture mutation steps executed in a sequence. Each pass can perform multiple iteration cycles including compilation, simulation and refining the processor architecture, as depicted in Fig. 2.

The overall idea of the exploration process is shown in Fig. 3. It follows an "expand and prune" approach: An initial unrealistically huge configuration is modified first by pruning underutilized parts and then *merging* such components which are utilized, but not commonly used *in parallel*, using analysis based on simulation data. The final result is an architecture that fulfills the goals set by the designer. The goal of the algorithm is to produce as efficient result (in terms of execution time) as possible while presenting the user with a few alternative implementations with different implementation characteristics to choose from.

All the traversed design space exploration points are stored in a database as processors configurations. From the set of configurations each pass generates, one or more are selected for the next stage. Selecting such configurations is crucial for the final processor architecture design as the ones likely leading to the most efficient desired end result ones must be selected. The phases in the proposed automated design space exploration algorithm are described in more detail in the following.

A. Starting Point

In the first pass, a processor architecture is created as a starting point. The resulting architecture is very large and not intended for implementation. However, it is a basis for the following stages which will incrementally prune resources that are not needed by the profiled program.

In this phase, all operations with known semantics are added as separate FUs supported by RFs of the required operand bit widths. Arithmetic and logical operations are added as individual ALUs, while for memory operations a separate load-store units (LSU) are created. In order to retain instruction level parallelism to the extent used by the intended number of FUs in the end result, ALUs and LSUs may be duplicated so that the same operation is available in more than one FU.

The operation latencies are set in this stage. For our FPGA implementation, operations have a default latency of two cycles in order to have a pipeline register before and after the operation logic, to isolate it from the IC network. Some more complex operations, namely bit shifts and multiplication, have a higher latency of 3 and 4 cycles, respectively.

In order to avoid register spilling at this stage, the RF sizes are set large enough to contain variables from any feasible program. A control unit and immediate unit are also added, and everything connected via single bus.

B. Operation Set Pruning

In the next pass the application is compiled for the huge architecture from the previous step and analyzed for the operation usage. FUs for operations that are not found in the compiled code are pruned off from the architecture. Since some expensive operations such as division, multiplication and floating point operations can be emulated in software, FUs can be reduced even further for such operations that are executed below a certain execution time threshold.

Since the goal is to always retain full compiler reprogrammability of the end result, a compiler required minimal set of operations such as addition, subtraction and basic memory operations are never pruned off for future successful application compilations.

C. VLIW Structure Generation

After pruning the FUs with underused operations, an architecture with a VLIW-like connectivity is created. In typical VLIWs, each FU input and output port are connected to a dedicated RF port of a corresponding width. Similarly, bypass

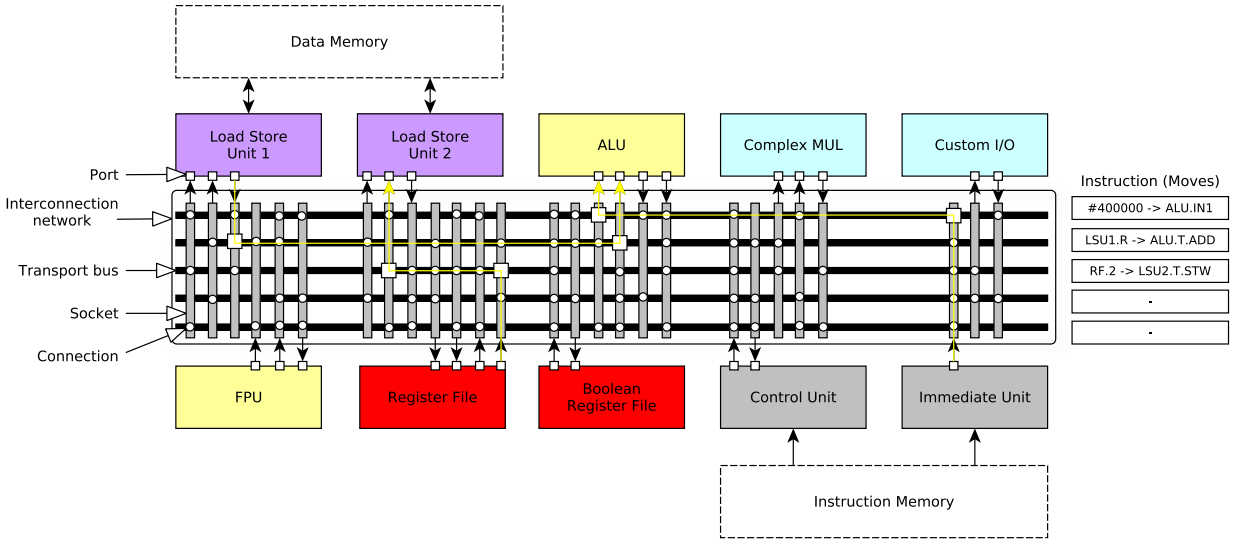

Fig. 1. Example of a TTA processor. In TTA, data transports between components are explicitly programmed. The example instruction defines an instruction with three moves controlling data transports in three buses out of the five. The instruction performs an integer summation of a value loaded from a data memory with a constant while simultaneously storing a previously computed value to memory.

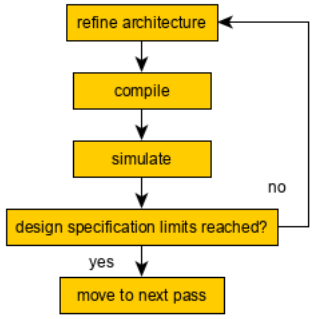

Fig. 2. Iterative process of a single exploration stage. The architecture is refined until the limitations specified by designer are met.

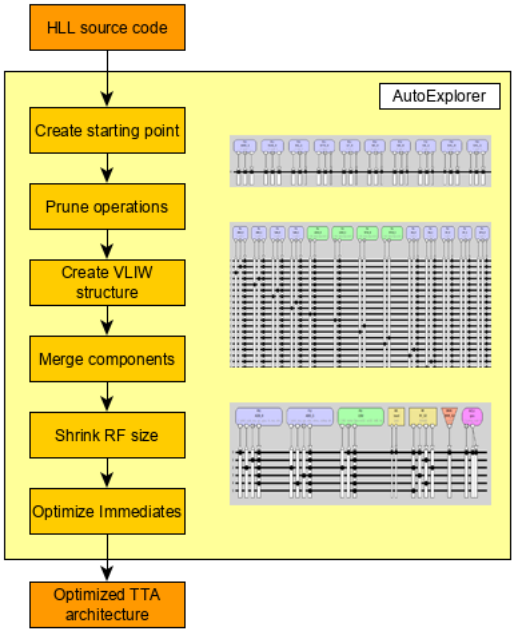

Fig. 3. Automated exploration and its passes. Snapshots on the right visualize the architecture's state in some of the stages; starting point, VLIW-connected and the final optimized architecture.

connections are added from all FU outputs to all FU inputs. This results in a general purpose, but unnecessarily complex IC network especially for the TTA programming model, which will be simplified in the next passes.

D. Component Merging

AEx reduces the number of parallel resources by merging them in case they are not typically utilized simultaneously. The types of merged resources include FUs, buses and RF ports. The benefit of merging the components is increased resource sharing that leads to more efficient utilization of the FPGA resources along with increased clock frequency that is affected especially by the reduced connectivity thanks to smaller number of components to be connected.

The basic principle behind the three merging phases is explained in more detail in [12]. In general, to efficiently merge two FUs, buses or RF ports, the simulation data containing utilization metrics is used to construct an *utilization covariance matrix*. Two components with the lowest covariance point out two resources that are rarely used simultaneously and thus can be merged without significant performance degradation via the resource sharing.

After this stage the architecture design point should be notably simpler than a general purpose VLIW structure produced in the previous stage.

E. Register File Shrinking

This phase shrinks the large RFs created at beginning until it starts to impact performance through too many variables spilling to memory. Also an optional RF splitting pass can be executed at this point to alleviate the RF complexity further as discussed in [11]. The idea is to split the RF to half in order to simplify each of the split RFs in terms of their port counts. The drawback of the RF splitting is the increased compiler duty of deciding which RF should hold which variable with minimal port conflicts.

F. Immediate Capability Optimization

The final two passes concern immediate encoding. The potential short immediate usage on each bus is profiled by starting with a 32b short immediate on each bus and compiling and simulating the program. Next, the short immediate widths are reduced until the average immediate width across buses goes below a certain threshold (6b by default) such that maximum number of the immediate moves seen in the simulation can still be done by short immediates.

The final phase creates a long immediate instruction template that is split over multiple buses. The choice of which buses to use for the immediate is not trivial, as those buses cannot be used when loading the immediate value. Therefore, if the long immediate is badly laid out, it can occupy slots that are used for moves that are in the critical path of the software, thus lengthening the program schedule. For this reason, simulation is used to determine the overall least used buses, which are used to their maximum width in the template to encode a 32b immediate.

V. Hardware Implementation Generation

The architecture exploration produces an *architecture*, that is, the programmer visible details of the processor. The architecture is defined as a file describing the components and their connections, but it does not yet define how they are *implemented* in hardware. Thus, an additional step is required to produce a synthesizable RTL for the soft core that can be utilized in the FPGA design. Fortunately, the modularity and the simplicity of the TTA microarchitecture makes hardware generation rather straightforward to automate which is not necessary to describe here as a whole.

The somewhat less trivial part is the generation of the FUs. Since AEx generates ALUs with arbitrary operation sets, a straightforward approach using premade FUs from an implementation database does not work. However, the LSU may be constrained to a model that can be premade, and does not need to be generated.

For this purpose, we developed an automated FU generator that can produce pipelined FUs with arbitrary operation sets out of simpler operations which are defined as *Operation RTL Snippets* (see Listing 1). The operations also contain semantical *Operation DAG* descriptions which describe the operation's semantics using one or more of other operations [13]. Thus, in order for the automated generation to work also for more complex operations, the operation snippets can be predefined for the most common basic operation set after which the FU generator can deduct implementations for any operation supported by means of expanding the Operation DAGs down to nodes that can be implemented.

Listing 1. Example Operation VHDL Snippet, performing a greater-than comparison.

```
if signed(op1) > signed(op2) then
  op3 <= '1';
else
  op3 <= '0';
end if;
```

For a number of basic operations, we wrote equivalent RTL implementations as short snippets of sequential statements, with the operands and results represented by placeholder signals. Most of these, like addition and bitwise logic operations, are one-liners, but the snippet can be as complex as needed.

The operation snippets are inserted into a synchronous process and executed when the operation is triggered. This limits them to an implementation with no registers. However, especially architecture-specific implementations may benefit from manual pipelining of complex operations such as multiplication. In order to support this, the function unit generator supports instantiating resources as components described to the tool through component definitions according to the IP-XACT standard [14].

These base operation implementations and the implementations generated from DAGs can be overridden with optimized or architecture-specific implementations. For example, we added an operation implementation for the 32-bit multiply and multiply-add operations that used hand-instantiated DSP blocks specific to the target architecture. This allows the automatic generation to be retargeted to different technologies and device families.

VI. Evaluation

The described automated exploration algorithm and the FU implementation generation was implemented on top of the open source application-specific instruction-set toolset *TTA-Based Co-Design Environment (TCE)* [15]. The result quality evaluation was made by specializing an architecture using AEx for the set of benchmark applications and comparing the results to architectures hand crafted in a previous publication [4].

While the proposed design space exploration algorithm does not have compiler algorithm specific parts as such, the compiler remains in a key role in the exploration process and, thus, in the quality of the results due to the software oriented processor architecture used. Therefore we start this evaluation section with a description of the extensions made to achieve the evaluation results.

A. The Compiler

The compiler used in the evaluations use the frontend and other architecture-independent parts from LLVM [5] version 6. When compiling for a new architecture, a dynamic library containing the architecture-specific parts is automatically built

and loaded as part of the compiler. Most essentially, the instruction scheduler used in the backend of the compiler also adapts to the processor architecture. The scheduling algorithm used in this evaluation is based on the one described in [16], which performs very aggressive software bypassing of values, avoiding many RF accesses. The algorithm was improved for this publication by adding operand sharing [17] to further reduce the pressure for register reads.

The register allocation of the compiler is performed in two phases: All the registers of the processors are introduced to the default register allocator of the LLVM compiler, which is run after the instruction selection of but before the instruction scheduling of the compiler. In case the architecture contains multiple register files of same width, which practically means that the register file splitting algorithm was ran, the registers in these register files are introduced to the LLVM register allocator in round-robin fashion between the register files. This makes the register allocator to use the different register files on a relatively even fashion, reducing the amount of register file port conflicts.

The early register allocation may cause antidependencies which can limit the instruction scheduling which is performed later. During the instruction scheduling the allocations which cause antidependencies that may limit the schedule are attempted to be re-allocated into registers which can eliminate antidependences to enable shorter schedules. However, not all antidependencies can be removed by this register reallocation, for example, if there is not enough registers available.

The exactly same compiler was used in the AEx exploration evaluation loop and when compiling for the reference architectures.

B. Results

The CHStone [18] benchmark suite was used for the application input. In this comparison, we chose the integer ones in order to clearly compare the results to the previously published results.

For each benchmark, two TTA soft cores were generated using AEx, with 2 or 3 FUs. Customized architectures were generated on Intel Core i5-6500 4-core platform running Ubuntu 16.04 operating system. Exploration times varied from 3 to 10 hours depending on benchmark and initial design parameters. For comparison, two of the TTA architectures were picked from [4]. As a high-performance point of comparison, we use *p-tta-3*, as it has the best execution time in every benchmark except *adpcm*. *m-tta-2* was chosen as a comparison for the resource-focused optimization point for its low LUT utilzation. The two MicroBlaze architectures from that paper – *mblaze-3* and *mblaze-5* – were used as additional resource- and performance-focused comparison points, respectively.

The synthesis methodology matched that of the hand-designed architectures used as the comparison. The soft processors were synthesized in out-of-context mode in Vivado with no area constraint for a synthesis target of a Zynq 7020 device with speed grade -1. The maximum frequency was found with a series of synthesis attempts using a binary search

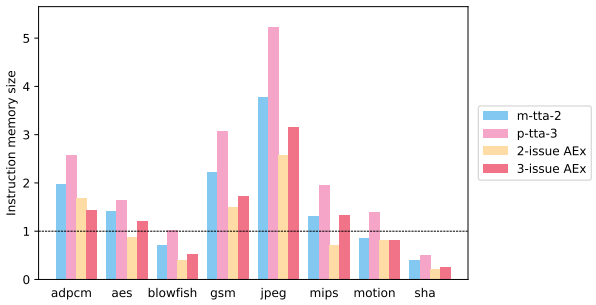

Fig. 4. Total instruction memory size relative to MicroBlaze.

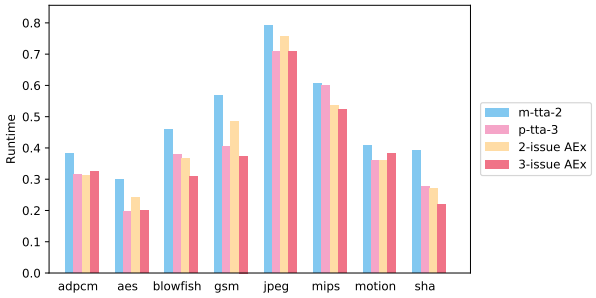

Fig. 5. Total runtime of high-performance architectures relative to *mblaze-5*.

of the target frequency. Synthesis times varied from 1 hour to 3 hours, depending on optimizations performed by Vivado. For the utilization figures, the machines synthesized at a clock frequency of 100 MHz without flattening the hierarchy to get accurate utilization divisions between components.

Table I summarizes the synthesis results for the compared architectures. The 2-issue AEx variants improve on the hand-made TTA variants in terms of look-up table (LUT) utilization, and in smaller architecture parallels *mblaze-5* resource utilization.

Notably, the 3-issue AEx architecture for *sha* has a significantly larger RF than any other, over four times as large as in *p-tta-3*. This is due to AEx selecting a RF with two write ports, which is inefficient to implement with the distributed RAM primitives of the used FPGA architecture, as they only have one write port. One solution would be to automatically split RFs that are too complex for the implementation technology, but this requires more architecture-specific parameters for the tool.

Additionally, the DSP block requirements are reduced: only *adpcm*, *gsm*, and *jpeg* use multiplication, and the remaining AEx architectures use no DSP blocks, while *m-tta-2* and *p-tta-3* use 3 and 6, respectively.

Overall, most AEx architectures reach maximum frequencies between 190 and 200 MHz, all above both of the tested MicroBlaze variants. However, none of them reach the clock frequencies available for *m-tta-2*.

Despite the lower clock frequency, both AEx architectures for each benchmark outperform *m-tta-2* as well as both MicroBlaze variants – *mblaze-3* saw worse performance in all

TABLE I
SYNTHESIS RESULTS FOR THE COMPARED ARCHITECTURES.

Architecture	LUT	IC	RF	FMax				
mblaze-3	715	0	128	169				
mblaze-5	829 (1.16x)	0	64	174 (1.03x)				
m-tta-2	1208 (1.69x)	437	44	212 (1.25x)				
p-tta-3	2651 (3.71x)	1290	72	197 (1.17x)				
	2-issue AEX				3-issue AEX			
Benchmark	LUT	IC	RF	FMax	LUT	IC	RF	FMax
adpcm	1091 (1.53x)	239	88	187 (1.11x)	1328 (1.86x)	255	88	177 (1.05x)
aes	929 (1.30x)	192	48	203 (1.20x)	1376 (1.92x)	343	72	193 (1.14x)
blowfish	895 (1.25x)	192	48	200 (1.18x)	1264 (1.77x)	304	48	198 (1.17x)
gsm	1029 (1.44x)	160	48	196 (1.16x)	1662 (2.32x)	336	72	187 (1.11x)
jpeg	1046 (1.46x)	176	48	191 (1.13x)	2067 (2.89x)	567	48	191 (1.13x)
mips	900 (1.26x)	192	48	195 (1.15x)	1328 (1.86x)	367	48	186 (1.10x)
motion	876 (1.23x)	223	48	192 (1.14x)	1051 (1.47x)	295	48	193 (1.14x)
sha	1039 (1.45x)	192	48	199 (1.18x)	1537 (2.15x)	319	306	183 (1.08x)

TABLE II
INSTRUCTION COUNTS AND WIDTHS AND THEIR RELATIVE SIZES IN COMPARISON TO [4].

	MicroBlaze		m-tta-2		p-tta-3		2-issue AEX		3-issue AEX	
Benchmark	IC	IW	IC	IW	IC	IW	IC	IW	IC	IW
adpcm	3918	32	3055	81	2416	134	2169	97	2194	82
aes	6483	32	3647	81	2560	134	2957	62	2542	99
blowfish	5665	32	1602	81	1396	134	1196	62	1053	89
gsm	3172	32	2778	81	2324	134	2451	62	1939	91
jpeg	9584	32	14318	81	11979	134	12768	62	11558	84
mips	1769	32	920	81	830	134	720	56	653	116
motion	3125	32	1070	81	1041	134	958	86	961	85
sha	6090	32	989	81	736	134	727	56	525	99

benchmarks than *mblaze-5*. 3-issue AEx results and *p-tta-3* are more even, with an average performance improvement by AEx of 6%. However, this is consistently achieved with a smaller LUT utilization, for up to a 60% reduction.

Another important factor in resource usage is the memory. While AEx cannot reduce the data memory usage, as that is intrinsic to the application, the processor architecture has an effect on the instruction memory. Table II summarizes instruction widths of the generated architectures and instruction counts of compiled applications for each of them.

Thanks to application-specific tailoring the instruction width is often narrower than the hand-designed architectures. This is due to fewer operations, more constrained scheduling freedom and optimized immediate encoding among others. Despite the less general interconnect, the instruction counts are also reduced.

The total instruction memory requirements are reduced as seen in Fig. 4: between m-tta-2 compared to *m-tta-2*, 2-issue AEx architectures require 5% to 49% less instruction memory, while the 3-issue AEx variants range from nearly even to a 35% reduction. As the TTA architectures use a different compiler from the GCC-based compiler for MicroBlaze, the instruction memory size varies significantly, but most of the 2-issue AEx variants require less memory than MicroBlaze.

VII. CONCLUSIONS

In this paper we introduced AEx, a tool for automated customization of exposed datapath soft-cores. Its purpose is to provide a HLS tool which presents a software programmable target and requires no FPGA specific knowledge. The customization results were compared to earlier hand made general purpose architectures published in [4]. The results showed that AEx can generate custom soft-cores which operate at similar or even better level of performance while providing significant resource savings.

In the future we plan to demonstrate AEx on a wide range of FPGA platforms from multiple vendors and perform detailed comparisons to commercial HLS tools. In terms of quality of results, more efficient automated utilization of SIMD instructions and special operations are expected to improve the results further. Also some effectiveness of the register file splitting could be improved by making the register reallocation phase of the compiler more aware of the register file port conflicts.

ACKNOWLEDGMENT

This research has been funded by Business Finland (FiDiPro Program funding decision 40142/14), the Academy of Finland (funding decision 297548), and ECSEL JU project FitOptiVis (project number 783162).

REFERENCES

[1] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? A study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[2] H. Ren, "A brief introduction on contemporary high-level synthesis," in *Proceedings of the 2014 IEEE International Conference on IC Design & Technology (ICICDT)*, May 2014.

[3] R. Nane *et al.*, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, Oct 2016.

[4] P. Jääskeläinen, A. Tervo, G. P. Vayá, T. Viitanen, N. Behmann, J. Takala, and H. Blume, "Transport-triggered soft cores," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018.

[5] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation Optimization*, Palo Alto, CA, Mar. 20–24, 2004.

[6] R. Nane, V. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels, "DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012.

[7] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011.

[8] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *23rd International Conference on Field Programmable Logic and Applications*, Sep. 2013.

[9] J. Hoogerbrugge and H. Corporaal, "Register file port requirements of Transport Triggered Architectures," in *Int. Symp. on Microarchitecture*, Nov.-Dec. 1994.

[10] P. Jääskeläinen, H. Kultala, T. Viitanen, and J. Takala, "Code density and energy efficiency of exposed datapath architectures," *J. Signal Process. Syst.*, vol. 80, no. 1, pp. 49–64, 2014.

[11] J. Janssen and H. Corporaal, "Partitioned register file for TTAs," in *Workshop on Microprogramming (MICRO-28)*, 1996.

[12] T. Viitanen, H. Kultala, P. Jääskeläinen, and J. Takala, "Heuristics for greedy transport triggered architecture interconnect exploration," in *Proc. Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. ACM, 2014.

[13] H. Kultala, P. Jääskeläinen, and J. Takala, "Operation set customization in retargetable compilers," in *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, Nov. 2011.

[14] "IEEE standard for IP-XACT, standard structure for packaging, integrating, and reusing IP within tools flows," *IEEE Std. 1685-2014*, 2014.

[15] P. Jääskeläinen, T. Viitanen, J. Takala, and H. Berg, "HW/SW co-design toolset for customization of exposed datapath processors," in *Computing Platforms for Software-Defined Radio*. Springer, 2016.

[16] H. O. Kultala, T. T. Viitanen, P. O. Jääskeläinen, and J. H. Takala, "Aggressively bypassing list scheduler for transport triggered architectures," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, July 2016.

[17] H. Kultala, J. Multanen, P. Jääskeläinen, T. Viitanen, and J. Takala, "Impact of operand sharing to the processor energy efficiency," in *Proc. CSI Int. Symp. Comput. Arch. Digital Syst.*, Tehran, Iran, Oct. 7–8, 2015.

[18] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *J. Inf. Process.*, vol. 17, 2009.