Customized Exposed Datapath Soft-Core Design Flow with Compiler Support

Otto Esko*, Pekka Jääskeläinen*, Pablo Huerta†, Carlos S. de La Lama†, Jarmo Takala*, and Jose Ignacio Martinez†
*Tampere University of Technology, Department of Computer Systems, Tampere, Finland
Email: {otto.esko,pekka.jaaskelainen,jarmo.takala}@tut.fi
†Universidad Rey Juan Carlos, Department of Computer Architecture, Móstoles, Madrid, Spain
Email: {pablo.huerta,carlos.delalama,joseignacio.martinez}@urjc.es

Abstract—**A popular way to exploit high level programming languages in FPGA designs is to use a soft-core with accompanying software development tools. However, a common shortcoming with the current soft-core offerings is their limited software execution capability: the required performance for the implementation can be often reached only with instruction set extensions.**

In this paper, we propose and evaluate an application-specific processor design toolset that uses a multi-issue exposed data path processor architecture template. The main benefit of the architecture is scalability with respect to instruction-level parallelism (ILP). The design flow allows the designer to freely customize the data path resources in the core to exploit the ILP available in computation intensive kernels. The design toolset includes a retargetable C compiler and an architecture simulator, making design space exploration feasible.

The experiments show that a relatively small soft-core tailored with the toolset provides significant speedups on software execution without using any instruction set extensions. The best measured speedup in comparison to the major commercial soft-cores was fourfold in applications from the CHStone benchmark suite, while the amount of consumed FPGA resources remained moderate.

I. Introduction

Custom intellectual property (IP) hardware blocks are often used for applications requiring high performance and low power consumption. However, manual IP block design process tends to be slow and error-prone. In addition, it often requires specific design skills and knowledge of hardware design. Methodologies and tools for high level synthesis (HLS) have gained interest in recent years due to the ever increasing complexity of new hardware designs [1]. The idea in HLS is to allow the application designer describe their application in a high level language (HLL) such as C, SystemC or Matlab while the HLS tool is responsible for mapping this description to hardware constructs as efficiently as possible. In addition to faster time-to-market, another advantage of such design methodologies is that they usually require less hardware design expertise, allowing people with software engineering skills to produce hardware IP with adequate performance.

A popular way of implementing high level synthesis for FPGAs is to use a soft-core processor based solution. The mainstream soft-core based programming of FPGAs has involved extending RISC-style soft-cores from the major FPGA vendors or cores available as open source with application-specific instruction extensions. A brief survey of the soft-cores can be found in [2]. The common shortcoming with the current soft-cores is the fixed-nature of the core with very limited support for customization. Scalar RISC cores are incapable of exploiting instruction-level parallelism (ILP), often available in computation intensive kernels. These cores resort mainly to instruction set extensions (ISE) and/or co-processors to achieve the required throughput, requiring substantial manual or algorithmic work in finding the beneficial ISEs.

In this paper, we propose and evaluate an application-specific processor (ASP) design flow that uses a multi-issue exposed data path processor architecture template. Main benefit of the architecture is its scalability with regards to instruction-level parallelism. The proposed design flow gives the freedom for the designer to customize the data path resources, such as function units and register files, to exploit the ILP available in the compiled software. In addition, the flow supports user defined custom operations with no limitations to the number of inputs and outputs nor the latency (thanks to the independent function unit pipelines).

The CHStone benchmarks evaluated in the paper show up to fourfold speedups in comparison to FPGA vendor soft-cores only by adding standard datapath resources to the designed architecture without using any instruction set extensions.

The rest of the paper is organized as follows. Section II discusses the related work on soft-core processors and toolsets used to customize them. Section III introduces the processor architecture template used in the proposed approach. Section IV presents the proposed ASP design flow, and Section V describes the integrated verification flow. The efficiency of the architecture and the toolset is evaluated in Section VI. Finally, section VII concludes the paper and outlines future directions.

II. Related Work

An ASP based on a VLIW soft core with customizable instructions is described in [3]. The work applies Nios II [4] instruction set on a 4-way VLIW architecture in order to exploit ILP. However, the degree of customization is limited; the core contains fixed set of resources in the data path and custom instructions are supported by adding units to the core.

CUSTARD is a customizable soft-core proposed in [5] and development tools include automated search for instruction set extensions (ISE). CUSTARD supports block and interleaved multithreading, which improves the throughput for multi-threaded programs. In our toolset, we support light weight block multithreading by means of compiler assisted context switches [6]. In the case of CUSTARD, the architecture template is again a simple MIPS-like RISC with custom operation I/O characteristics limited by MIPS instruction encoding. In the proposed approach, there are no limitations to the number of inputs or outputs nor internal pipelining of the custom operations.

Several HLS tools generating RTL from high level languages have been commercialized, e.g., Synopsys recently announced an HLS flow starting from Matlab programs [7]. Our design flow can be considered to belong to the category of HLS tools as it provides a toolset for supporting gradual conversion of HLL programs to a customized static multi issue processor and automatically parallelized code.

A similar architecture and a toolset to ours is presented in [8] where the designer can customize the number and type of FUs and RFs in the data path. Clustered VLIW is used to avoid the register file port bottleneck inherent to VLIWs. In addition, their forwarding network is customizable to reduce the interconnection network complexity. However, an important and the most complex part of the toolset is missing; there is no retargetable HLL compiler. The generated processors can only be manually assembly programmed, making design space exploration practically impossible.

Finally, the proposed toolset includes a retargetable C compiler unlike many other ASP based design toolsets. In addition, our toolset has been tested with several large applications and has a fully functional and stable open source release available for download at [9].

III. Transport Triggered Architectures

VLIWs are considered interesting for applications with high processing performance requirements [10] and relatively simple control flow that can be predicated. Transport Triggered Architecture (TTA) is a modular processor architecture template with similarity to VLIW architectures [11]. TTA can be described as an "exposed data path VLIW": instead of defining which operations are started in which function units (FU) at which instruction cycles, TTA programs are defined as data transports between register files (RF) and FUs of the data path. The operations are started as side-effects of writing operand data to the "triggering port" of the FU. The modularity of TTA allows easy customization of processor designs, making it an interesting architecture template for automated processor generation. Fig. 1 presents a simple example of a TTA processor.

The programming model of VLIW imposes limitations for scaling the number of FUs in the data path. Increasing the number of FUs has been problematic in VLIWs due to the need to include as many write and read ports in the RFs as there are FU operations potentially completed and started at the same

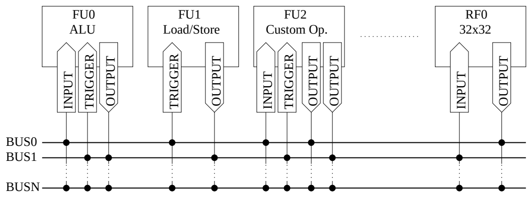

Fig. 1. Example of a TTA processor.

time. Additional ports increase the RF complexity, resulting in larger area, critical path delay and power consumption. The increased RF complexity was perceived as the main limitation for maximum clock frequency for a VLIW on an FPGA in [3] and was addressed by a clustered VLIW approach in [8]. Thanks to its programmer-visible interconnection network, TTA data path can support more FUs with simpler RFs [12]. Because the scheduling of data transports between data path units are programmer-defined, there is no obligation to scale the number of RF ports according to the number of FUs [13]. Using multiple simpler RFs instead of a monolithic complex one is a common design choice for TTA designers.

The second main benefit of TTAs compared to VLIWs is the customizable interconnection network. Adding an FU to the VLIW data path requires potentially new bypassing paths to be added from the FU's output ports to the input ports of the other FUs in the data path, which increases the interconnection network complexity. In addition to the register file bypasses, the whole data path connectivity can be tailored according to the application at hand, adding only the connectivity that benefit the application the most. The customization of the interconnection network combined with the possibility to program data transports directly from an function unit to another (a.k.a. software bypassing) can lead to notable performance increases [14].

One interesting aspect of TTA is the independence of its FU pipelines and their independence from the RFs. While in operation-triggered architectures FUs are tightly coupled to the connected RFs, in case of TTAs there is no such coupling thanks to the transport programming. This allows TTAs to integrate also longer latency co-processors directly to the data path using the regular FU port interface [15] and it allows more scheduling freedom for the compiler.

IV. From C to ASP on FPGA

This section describes the proposed design methodology for converting C programs to ASPs running parallelized programs on FPGA (see Fig. 2). The first subsection describes the basic design flow in which only "basic processor resources" are customized, the second subsection describes the steps needed to use instruction set extensions, and finally the third subsection describes the most important tool: the retargetable C compiler.

A. Basic Design Flow

The design flow starts with a HLL source code and an initial processor architecture definition file (ADF). The initial

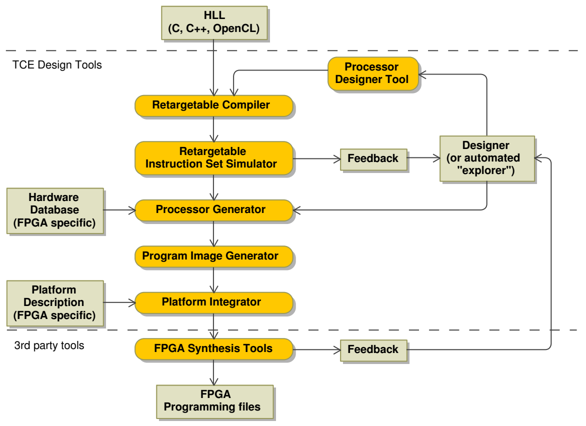

Fig. 2. TCE design flow from high level language to FPGA.

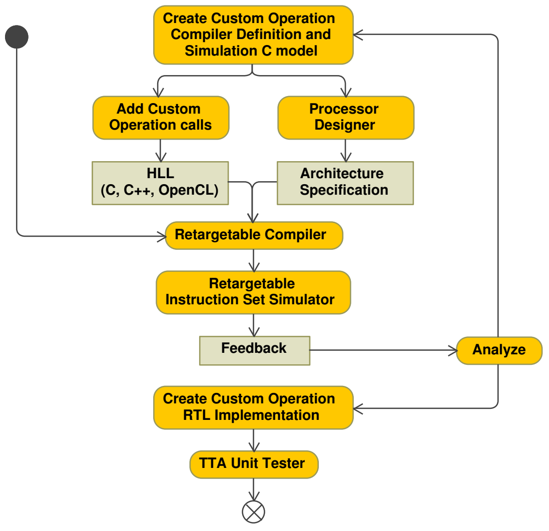
Create Custom Operation Compiler Definition and Simulation C model
Add Custom Operation calls
Processor Designer
HLL (C, C++, OpenCL)
Architecture Specification
Retargetable Compiler
Retargetable Instruction Set Simulator
Feedback
Analyze
Create Custom Operation RTL Implementation
TTA Unit Tester

Fig. 3. Using custom operations.

architecture is usually a processor architecture with a minimal set of resources that still allow C compilation or another previously designed generic architecture.

The HLL source code and the ADF are given to the compiler which produces an assembly program. The compiler is runtime retargetable thus it adapts automatically to the set of resources defined in the ADF. The main task of the compiler is to optimize the input and parallelize it at the instruction level to exploit the given architecture as efficiently as possible.

The next step is to simulate the program using a retargetable cycle accurate processor architecture simulator. The simulator produces profiling info, such as the cycle count and resource utilization. This feedback is directed back to the designer or, if fully automated design flow is desired, to an automated architecture design space exploration tool. Based on the feedback, the architecture is customized by adding or removing FUs, RFs or data path connectivity. Manual customization is usually done using the Processor Designer tool with a graphical user interface. The program is recompiled to the modified architecture and the simulator feedback is again analyzed. After a satisfactory cycle count is reached, compile-simulate-customize cycle is finished and the designer proceeds to processor HDL generation.

At the HDL generation phase, the designer first chooses implementations for the processor components specified in the ADF. The FU and RF RTL implementations are stored in Hardware Databases (HDB) along with their implementation data. HDBs are libraries of processor components which can be target platform specific or generic. FPGA-specific HDBs contain implementations tailored for a certain FPGA platform. Tailoring is useful, for example, to access the I/O on the FPGA board through an FU interface, or to implement an RF that uses FPGA's internal memories optimally. HDBs can also store cost data of the implementations that is used by the Cost Estimator tool (excluded from Fig. 2 for simplification) to estimate the area, maximum clock frequency, and energy consumption of the processor implementation before the logic synthesis.

After the architectural components in ADFs are bound to implementations in HDBs, the processor RTL implementation can be produced using the Processor Generator tool. This tool collects the chosen FU and RF implementations and generates the interconnection network and the control unit. The control unit includes the instruction decoder, optional decompressor (in case instruction compression is used), and the fetch unit for loading the instruction words from the instruction memory. In addition, it creates a generic test bench that can be used in HDL verification.

The next phase is to generate the instruction and data memory bit images of the program executable. This is done by using Program Image Generator which generates the encoding for the instructions and supports multiple output formats, thus the designer can choose the most appropriate one for his platform.

The last tool in the TCE FPGA design flow is the Platform Integrator. The main purpose of this tool is to connect memory components and other external IPs to the TTA core. In addition, the tool creates project files for 3rd party synthesis tools and performs I/O signal mapping to physical FPGA pins. The final steps are synthesis, place, and route of the processor which are performed using 3rd party tools.

B. Custom Operations

In addition to tailoring the processor architecture with "basic data path resources", the designer can also create user-specific instruction set extensions (custom operations) to further accelerate the application at hand. Fig. 3 illustrates the phases in evaluating and using custom operations. In the current toolset version, automatic ISE search is not supported yet. However, we have found that the best ISEs are usually found manually by exploiting a priori knowledge on the algorithmic, instead of trying to find extension candidates by automatic scanning of low level program representations.

After a suitable custom operation candidate is found (by using program profile data, for example) the designer needs to create a compiler definition for the operation. This definition tells how many input and output operands the operation has and whether the operation has any special features such as internal state or if it accesses memory. In order to simulate the operation in the architecture simulator, a C/C++ simulation model, also known as a behavioral model of the operation, is defined. This model can be usually copy-pasted directly from the accelerated part of the input program with minor modifications.

The next step is to add an FU which implements the new operation to the processor architecture. In order to use the custom operation from the program, the designer needs to insert custom operation calls into appropriate places of the source code. After inserting the custom operation calls and an FU supporting the custom operation to the architecture, a new compile-simulate-analyze iteration begins. From the analysis, the designer decides either to try another custom operation candidate or to proceed to implementation.

It should be noted that only after the designer has found the appropriate set of custom operations and reached the desired cycle count, it is time to create the RTL implementations of the custom operations. Using high level descriptions of the operations and C simulation models makes testing custom operation candidates easy and fast as the hardware implementations of the operations are not needed until the processor will be implemented. It should also be noted that the each function unit has a standard interface, thus the RTL description of a custom operation is not a complex task.

The implementation of the custom operations in VHDL is the only manual HDL writing step required at the moment in the TCE design flow. The implemented special function units are stored in HDBs which can be reused in the future designs, potentially removing even this implementation step for new processor designs.

C. Retargetable C Compiler

As TTA is a statically scheduled architecture with low level details of execution exposed to the programmer, the runtime efficiency of the end results produced with the design toolset depends heavily on the quality of the compiler.

TCE uses the LLVM Compiler Infrastructure [16] in its compiler tool chain (later referred to as 'TCECC'), thus benefits from its global optimizations, such as aggressive dead code elimination and link time inlining. LLVM provides the frontend, the middle end optimizations, and parts of the backend. The final phases of TCECC code generation have been written from the scratch to provide efficient retargetable instruction scheduling and TTA-specific optimizations.

V. Verification Support

Although the proposed ASP design flow is automated, it is often useful to be able to verify the implementation at the different stages of the design flow. This section describes the main ways to verify the generated processor designs: top-down verification and processor unit testing.

A. Top-Down Verification

The basic idea in top-down verification is to compare the verification outputs with the ones produced from a previous stage in the design flow, essentially isolating the point in the design flow that introduced the detected failure. Top-down verification is further divided to two different methods depending on the type of output used as the verification data: standard output or bus traces.

1) Standard Output: This method relies on C *stdio.h* functions such as *printf()*, *puts()*, and *putc()* to produce verification data from the input program itself. The output can be produced on four different stages in the design flow:

1) Native workstation execution. HLL program is compiled and ran natively on the designer's workstation. This usually produces the known correct output which is used as the comparison reference for the rest of the stages.
2) Architecture simulation. The HLL program is compiled with TCECC and executed using the architecture simulator. TCECC *stdio* implementation uses a custom operation called "STDOUT" to output characters. The default simulation model outputs the chars to the simulator's console.
3) RTL simulation. The FU containing the STDOUT is implemented at this stage using the simulation-only printing functions of VHDL.
4) FPGA execution. Unlike in the previous stage, now the STDOUT implementation needs to be synthesizable. Usually the implementation uses JTAG or UART to output the characters to a console or a display.

2) Bus Trace: A bus trace includes the values in every transport bus (the buses controlled by TTA programs) at every clock cycle. The bus traces can be obtained from three different stages in the design flow:

1) Architecture simulation. This bus trace is used as the reference for later comparisons.
2) RTL simulation. When generating the processor HDL designer can enable an option which creates a bus trace recording module to the processor's interconnection network. This module is implemented using the VHDL file I/O API, thus is not synthesizable.
3) FPGA execution. Implementation of the bus trace recording module in the decoder is changed to a synthesizable design specific to the used FPGA platform. Using JTAG is an obvious choice for the output.

B. Processor Unit Testing

Processor unit testing enables automated FU and RF implementation testing. In essence, the idea is to introduce the test-driven development (TDD) workflow [17], popular in agile software development, to the processor design flow. Using processor unit testing, the designer can first create only the architecture and the C simulation model for a new FU (usually for a custom operation) and then start implementing the RTL version until its unit test passes.

Unit testing is implemented with two programs: the TTA unit test generator, which is included in TCE toolset, and a

3rd party RTL simulator. The test generator takes a processor implementation description as an input. This file defines all the FU and RF implementations used in the architecture.

In order to create reference output data automatically, the unit test generator uses the architecture simulation models of the tested FUs. Input vectors for the tested FUs are generated (currently randomly) and the FU simulation models are used to generate the reference output data vectors. An HDL testbench is created using the test vectors that feeds input data to the FUs and compares the produced output to the reference. If the data differs the testbench issues VHDL assertions. After the RTL testbench is generated, the unit tester uses an RTL simulator to simulate the testbench and informs the user about the errors found.

VI. Benchmarks

In order to measure the efficiency of the proposed design methodology, toolset and the associated processor template against existing soft-core implementations, a set of applications were implemented using a TTA designed with the toolset. No instruction set extensions were used in order to compare strictly the software execution capabilities of the cores. The machine used was partially connected, with three *cluster* nodes comprising one ALU and one register file each, and a global bus for direct data transports between different clusters. ALU operations had a latency of 2 cycles.

Comparison was done against a Nios II/f processor, synthesized on an Altera Stratix II FPGA, and two different MicroBlaze configurations, with 3- and 5-stage pipelines, running on a Xilinx Virtex 5 family FPGA. In order to avoid the effect of different arithmetic emulation libraries, we used hardware multipliers on all the targets. On the TTA machine, a multiplier with 4 cycle latency was chosen to increase the maximum clock frequency.

Test applications were taken from the CHStone benchmarking suite [18], targeted for measuring the efficiency of HLS tools. Due to the current lack of 64-bit integer support in our toolset, some of the tests provided by CHStone had to be excluded from the measurements.

Table I summarizes execution times for the test cases on each target, as well as maximum achievable frequency for each platform. It has to be noted that GSM test produced incorrect results on MicroBlaze, due to unknown causes (our guess is a compiler error), and was therefore removed from the analysis. Taking these data into account, performance results are given in Fig. 4, which shows the speedup of the designed TTA soft-core against the compared soft-cores.

The worst speedup is produced with the MIPS test, with nearly two-fold improvement over the MicroBlaze cores but 25% slowdown in comparison to the Nios II/f processors. This is because the MIPS test is composed of a large number of small basic blocks. This ruins the possibility to extract parallelism from the algorithm, and also causes very frequent conditional branches with very few instructions in those conditional blocks to fill delay slots with. Nios II/f cores have a built-in dynamic branch predictor which allows them to outperform other targets in such control-oriented applications.

TABLE I
Execution time (μsec) and maximum frequency (MHz) for each test platform.

Test	TTA on Stratix II	Nios II/f	TTA on Virtex 5	3-stage mBlaze	5-stage mBlaze
adpcm	542.0	691.8	422.8	1 098.9	801.5
aes	176.7	396.2	137.8	469.3	376.6
blowfish	5 754.6	7 285.0	4 489.2	11 192.8	8 571.7
gsm	118.5	133.4	92.4	n/a	n/a
jpeg	46 826.4	120 817.1	36 529.5	142 344.0	118 529.2
mips	259.6	187.0	202.5	389.2	377.7
sha	3 176.1	4 054.0	2 477.7	10 016.5	7 658.3
$f_{\max}$	149	175	191	169	195

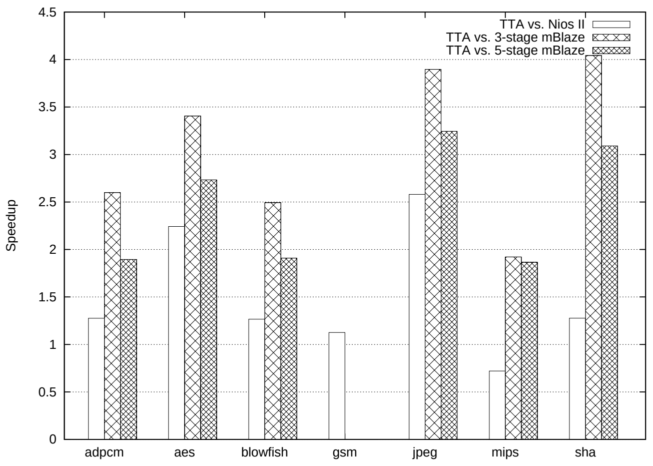

Fig. 4. Speedup of the designed TTA over FPGA vendor soft-core processors.

The best cases are JPEG and AES with about fourfold speedups. The analysis shows that these algorithms consist of the largest number of arithmetic and logic operations, and after unrolling, predication and inlining end up composed of large unconditional basic blocks, thus providing better utilization for the multi-ALU TTA core.

The comparison of FPGA resource usage is given in table II. These numbers show that the area utilization of the TTA core is roughly three times larger compared to the reference soft-core processors. This is a moderate increase in FPGA utilization when considering the significant speedups and the fact that these speedups were produced merely by using basic TTA resources and no custom operations. In addition, it is important to notice that in these benchmarks we used a single generic TTA tailored towards the demands of the most computation intensive programs in the benchmark set. In case better performance/resource consumption ratio is desired for each separate program, it would be trivial to simplify the TTA by removing datapath resources for the cases with lower computational demands.

Naturally, for a VLIW-like architecture such as TTA, one significant aspect is the instruction memory usage. Due to the need to encode NOPs, the width of a TTA instruction tends to be big in comparison to the scalar RISC archi-

TABLE II
SYNTHESIS RESULTS FOR ALTERA STRATIX II AND XILINX VIRTEX 5 FPGAS.

		Stratix II		Virtex 5		
		TTA	Nios II	TTA	mBlaze (3 stage)	mBlaze (5 stage)
LUTs	#	5 218	2 322	5 024	1 537	1 889
	% of all	3.6 %	1.6 %	7.3 %	2.2 %	2.7 %
Registers	#	2 785	1 896	3 485	1 318	1 841
	% of all	1.9 %	1.3 %	5.0 %	1.9 %	2.7 %

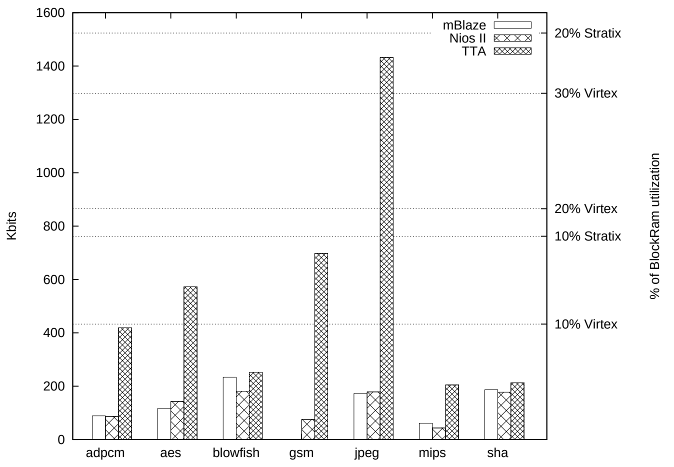

Fig. 5. Instruction memory usage of MicroBlaze, Nios II/f and TTA without instruction compression

tectures. Instruction memory requirements of the benchmark applications are presented in Fig. 5. Although the instruction memory usage of TTA is notably larger on some applications, the total memory usage is still moderate compared to the amount of available memory in the current FPGAs. In case instruction memory usage becomes a problem, TCE supports using instruction compression which can drastically lower the TTA instruction memory size. However, when compression is used, the instruction decompressor module, including the dictionary in case of dictionary compression, increases the FPGA resource usage. Therefore, it depends on the type of available FPGA resources when instruction compression is feasible.

VII. Conclusion and Future Work

We have proposed an application-specific processor (ASP) design flow based on the use of an exposed data path architecture. The main advantages of the design flow are its runtime retargetable C/C++ compiler, free customization of the data path resources including the connectivity, and the easy verification of the designs.

In the CHStone benchmarks, a simple soft-core produced with the toolset outperformed implementations based on soft-core processors provided by the major FPGA vendors, even without the use of any instruction set extensions tailored for each application. The measured FPGA resource usage increase was moderate compared to the achieved increase in performance.

In the future we plan to implement algorithms for automatic instruction set extension search, improve the instruction scheduler's efficiency with sparsely connected machines, and add support for generating multicore ASPs.

Acknowledgment

This research was partially funded by the Academy of Finland. The 2nd author thanks the Foundation of Nokia Corporation for financial support.

References

[1] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *Design & Test of Computers, IEEE*, vol. 26, no. 4, pp. 18–25, July-Aug. 2009.

[2] J. Tong, I. Anderson, and M. Khalid, "Soft-core processors for embedded systems," in *Proc. Int. Conf. Microelectronics*, Dhahran, Saudi Arabia, Dec. 16–19 2006.

[3] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, "An FPGA-based VLIW processor with custom hardware execution," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Monterey, CA, USA, 2005, pp. 107–117.

[4] Altera Corp., "Nios II processor reference handbook," 2009.

[5] R. Diamon, O. Mencer, and W. Luk, "Application-specific customisation of multi-threaded soft processors," *IEE Proc. - Comput. Digit. Tech.*, vol. 153, no. 3, pp. 173–180, May 2006.

[6] P. Jääskeläinen, P. Kellomäki, J. Takala, H. Kultala, and M. Lepistö, "Reducing context switch overhead with compiler-assisted threading," in *Proc. IEEE/IFIP Int. Conf. Embedded and Ubiquitous Computing*, Shanghai, China, Dec. 17–20 2008, pp. 461–466.

[7] C. Eddington, "Synthesizing algorithms from MATLAB and model-based descriptions: Introduction to Synphony HLS," Synopsys, Inc., Tech. Rep., 2009.

[8] M. A. R. Saghir, M. El-Majzoub, and P. Akl, "Datapath and ISA customization for soft VLIW processors," in *Proc. IEEE Int. Conf. Reconfigurable Comput. FPGA's*, San Luis Potosi, Mexico, Sept. 20–22 2006, pp. 1–10.

[9] "TCE: TTA-based codesign environment." [Online]. Available: http://tce.cs.tut.fi

[10] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Proc. Int. Conf. Architectural Support for Programming Languages Operating Syst.*, Palo Alto, CA, Oct. 5–8 1987, pp. 180–192.

[11] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*. Chichester, UK: John Wiley & Sons, 1997.

[12] ——, "TTAs: missing the ILP complexity wall," *J. Syst. Architecture*, vol. 45, no. 12-13, pp. 949–973, 1999.

[13] J. Hoogerbrugge and H. Corporaal, "Register file port requirements of Transport Triggered Architectures," in *Proc. Annual Int. Symp. Microarchitecture*, San Jose, CA, Nov. 30–Dec. 2 1994, pp. 191–195.

[14] V. Guzma, P. Jääskeläinen, P. Kellomäki, and J. Takala, "Impact of software bypassing on instruction level parallelism and register file traffic," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. Lecture Notes in Computer Science, M. Bereković, N. Dimopoulos, and S. Wong, Eds. Heidelberg, Germany: Springer, 2008, vol. 5114, pp. 23–32.

[15] P. Jääskeläinen, H. Kultala, T. Pitkänen, and J. Takala, "Reducing the overheads of hardware acceleration through datapath integration," in *Proc. SPIE Multimedia on Mobile Devices*, vol. 6821, Jan. 27–31 2008, pp. 6821R–1 – 10.

[16] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation Optimization*, Palo Alto, CA, March 20–24 2004, p. 75.

[17] K. Beck, *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[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," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.