DWARV 2.0: A CoSy-based C-to-VHDL Hardware Compiler

Razvan Nane*, Vlad-Mihai Sima*, Bryan Olivier†, Roel Meeuws*, Yana Yankova* and Koen Bertels*
*Delft University of Technology
Email: ⟨r.nane, v.m.sima, r.j.meeuws, y.d.yankova, k.l.m.bertels⟩@tudelft.nl
†ACE Associated Compiler Experts bv
Email: bryan@ace.nl

***Abstract*—In the last decade, a considerable amount of effort was spent on raising the implementation level of hardware systems by automatically extracting the parallelism from input applications and using tools to generate Hardware/Software co-design solutions. However, the tools developed thus far either focus on particular application domains or they impose severe restrictions on the input language. In this paper, we present the DWARV 2.0 compiler that accepts general C-code as input and generates synthesizable VHDL for unrestricted application domains. Dissimilar to previous hardware compilers, this implementation is based on CoSy compiler framework. This allowed us to build a highly modular compiler in which standard or custom optimizations can be easily integrated. Validation experiments showed speed-ups of up to 4.41x when comparing against another state of the art hardware compiler.**

***Keywords*-DWARV; HW Compiler; FSM; C-to-VHDL;**

I. INTRODUCTION

Even though hardware compilers which take as input a High Level Language (HLL) and generate a Hardware Description Language (HDL) are no longer seen as exotic technology, they cannot yet be seen as a mature technology to the same extent as e.g. software compilers. Hardware compilers are especially used to develop application specific hardware where for various application domains the computational intensive part(s) are accelerated. They are a vital component of the Hardware/Software (HW/SW) co-design effort needed when FPGA based kernels are involved. In this paper, we specifically look at FPGA based platforms where parts of the application will stay on the General Purpose Processor (GPP) and other parts will be transformed into Custom Computing Unit (CCU). To perform fast Design Space Exploration (DSE), it is necessary to evaluate the different mappings of the application, with their corresponding HDL implementation, on the hardware platform. To this purpose, hardware compilers allow the designer to immediately obtain a hardware implementation and skip the manual and iterative development cycle altogether.

However, current hardware compilers suffer from the lack of generality in the sense that they support only a subset of a HLL, for example no pointers or Floating Point (FP) operations accepted. Even more, only a few allow the application programmer to use other function calls inside the kernel (i.e. unit) function. This leads to manual intervention to transform the input code to a syntax accepted by the compiler, which is both time consuming and error prone. These problems are caused by the fact that hardware generators are typically bound to one particular application domain or are implemented in compiler frameworks that provide cumbersome ways of generating and processing the Intermediate Representation (IR) of the input code. Our contribution is threefold:

- Provide a redesign of the DWARV hardware compiler [2] using the CoSy compiler framework [8] to increase the coverage of the accepted C-language constructs.
- Provide a general template for describing external Intelectual Property (IP) blocks, which can be searched and used from an IP library to allow custom function calls.
- Validate and demonstrate the performance of the DWARV 2.0 compiler against another state of the art research compiler. We show kernel-wise performance improvements up to 4.41x compared to LegUp compiler [3].

The rest of the paper is structured as follows. In Section II we present an overview of existing HDL generators. Section III gives details about the compiler tool-flow and the template descriptor used for external IP blocks supporting custom function calls. Section IV validates DWARV 2.0 by presenting the comparison results, while Section V draws the conclusion.

II. PREVIOUS WORK

Plenty of research projects addressed the issues of automated HDL generation. The ROCCC project [4] aims at the parallelization and acceleration of loops. Catapult-C [6] and CtoS [7] are commercial high-level synthesis tools that take as input ANSI C/C++ and SystemC inputs and generate register transfer level (RTL) code. The optimizations set of the SPARK [5] compiler is beneficial only for control-dominated code, where they try to increase the instruction level parallelism. In addition, the explicit specification of the available hardware resources such as adders, multipliers, etc. is required. In contrast to these compilers, DWARV 2.0 does not restrict the application domain and it is able to generate hardware for both streaming and control intensive applications. Furthermore, it does not restrict the accepted input language. DWARV 2.0 allows a large set of C constructs including pointers and memory accesses. Finally, no additional user input is necessary.

Altium's C to Hardware (CHC) [11], LegUp [3] and DWARV 2.0's predecessor [2] are the compilers that resemble the closest to DWARV 2.0. They are intended to compile annotated functions that belong to the application's computational

intensive parts in a HW/SW co-design environment (although the latter can compile the complete application to hardware as well). They are therefore intended to generate accelerators for particular functions and not autonomous systems. This is typical for Reconfigurable Computing (RC) Systems and the same assumption is true for DWARV 2.0 as well. However, there are also two major differences, the IP reuse and the more robust underlying framework. The first feature allows custom function calls from the HLL code to be mapped to external IP blocks provided they are available in external IP libraries. The second feature enables seamless integration of standard or custom optimization passes.

III. DWARV 2.0

In this section, we describe the DWARV 2.0 compiler by highlighting the improved aspects comparing with the previous version. We present the engine flow, the new features and describe the IP library support.

A. DWARV 2.0 *Engines: The Tool-Flow*

DWARV 2.0 targets reconfigurable architectures following the Molen [1] machine organization and is built with CoSy [8]. Compilers built with CoSy are composed of a set of *engines* which work on the IR of the input program. The initial IR is generated by the C front-end, which is a standard framework engine. To generate VHDL from C code, DWARV 2.0 performs standard and custom transformations to the combined Control Data Flow Graph (CDFG) created in the IR by the *CFront* engine. Figure 1 depicts this process graphically, highlighting on the left side the three main processing activities required for C-to-VHDL translation. On the right side of the same figure, we show in clock-wise order an excerpt of the most important engines used in each activity box shown on the left side.

The *CFront* (ANSI/ISO C front end) creates the IR. The *cse* and *ssa* engines perform common sub-expression elimination and static single assignment transformations. The *match* engine creates rule objects by matching identified tree patterns in the IR, while the **psrequiv** engine annotates which register/variable actually needs to be defined in VHDL. **fplib** searches and instantiates HW templates found in the library. **hwconfig** reads in parametrizable platform parameters, e.g. memory latency. **setlatency** places dependencies on def/use chains for registers used by IP cores. It sets the latencies on memory dependencies as well. *sched* schedules the CDFG and *dump* prints IR debug information. Finally, the **emit** engine emits IEEE 754 synthesizable VHDL. The engines given in bold in Figure 1 are custom and thus written specifically for VHDL generation. The rest are standard framework engines. Because a total of 52 (43 standard - 9 custom) engines were used in the design of DWARV 2.0 and considering space limitations, we did not give the complete list of engines used.

B. *New Features and Restrictions*

Table I summarizes DWARV 2.0's new features. Leveraging the availability of generic lowering engines, which transform specific constructs to basic IR operations, most of the previous

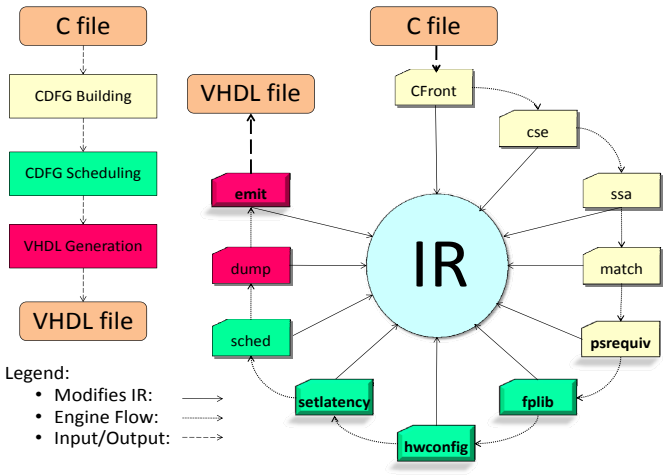

Fig. 1: DWARV 2.0 engines. Clock-wise sequential execution of engines starting from CFront.

TABLE I: DWARV 2.0 New Features.

Data Types	Statements	IP Library Fields
Integer 64 bit	div, mod	IP name
Real Floating Point	case, label, switch	input ports names
Multi-dimensional arrays	function calls	output port names
Struct	while, do-while	operation type & size
Union	return, break	latency & frequency

syntax restrictions were removed. The best example is the support for structured aggregate data types. Another major development was the FP and the template library. This not only facilitates the addition of FP operations, but provides also a generic mechanism to support function calls.

To add support for the basic FP arithmetic, we first use the Xilinx tool *coregen* to generate FP cores (e.g. for multiplication). We describe then these generated IP cores in a library that DWARV 2.0 is able to search for an appropriate core for each of the floating point operations. Table I third column lists the important fields that DWARV 2.0 must know in order to find the proper core, instantiate and schedule it in the design. The same syntax can be used to describe and support generic function calls as well. The only exception is that for the operation field name, instead of using an *operation type* identifier, we simply use the function name.

Although the new version provides new features, there are still some restrictions. The first two restrictions are related to the fact that there is no stack on an FPGA. This implies that functions can not be recursive and that static data is not supported. Implementing a stack would be possible, but would defeat the purpose of hardware execution because it will limit the available parallelism. The third restriction is that mathematical functions present in the standard C library are not available. This restriction could be lifted in the future, using the function call support described.

IV. EXPERIMENTAL RESULTS

To assess the performance of DWARV 2.0, we compared cycle, frequency and area information obtained by generating

and simulating the CCU hardware for eight kernels against the hardware IP produced by the LegUp compiler from Toronto University [3]. In this section, we briefly describe the LegUp compiler, the platform and the comparison experiments.

LegUp Compiler

LegUp [3] is a research compiler developed at Toronto University which was developed using LLVM [9]. It accepts standard C-language as input, and generates Verilog code for the selected input functions. Its main strength is that it can generate hardware for complete applications or only for specific application functions, i.e. accelerators. In this latter case, a TigerMIPS soft processor [10] is then used to execute the remainder of the application in software. The connection between these two main components is made through an Avalon system bus. This is similar to the Molen machine organization, therefore comparing the execution times of accelerators generated by this tool is relevant to assess the performance and development state of DWARV 2.0. LegUp was reported [3] to perform close to an industrial HLS compiler, i.e. eXCite [12], which, assuming transitivity of results, was another reason to use LegUp as our benchmark.

Experimental Platform

To compare the DWARV 2.0 and LegUp compilers, we followed a two step approach. First, we simulated the generated kernels to obtain the cycle information. The simulation infrastructure for DWARV 2.0 is designed in such a way to return only the execution time for the individual kernel invocation. However, for the LegUp simulation, care has to be taken to obtain only the execution time for the kernel itself and not for the complete test-bench as it is when the hybrid execution is chosen. To obtain the correct number, the ModelSim wave-form had to be opened and the difference between the start and finish signals had to be computed.

Subsequently, we ran a full post-place and route synthesis to obtain the maximum frequency and area numbers for the Xilinx Virtex5 ML510 development board. To obtain a meaningful comparison, we needed to integrate the kernels generated by LegUp in the Molen work-flow to target the same board. To this purpose, we wrote wrappers around the LegUp interface. Note that these wrappers do not influence the performance comparison. We use these wrappers only for integration purposes to be able to target a different platform than the one for which LegUp kernels were generated. Doing so, we are interested only in the area numbers obtained for Xilinx instead of Altera board. The performance numbers are computed thus in the original setting without any wrappers included. Given that the tools target similar heterogeneous platforms with the accelerated kernels running on the Xilinx board as co-processors of the PowerPC processor vs. Altera/TigerMIPS platform, the mismatch in interfaces was minimal and easy to correct. Both interfaces contained ports to start the accelerator, query its status and read/write data from the shared memory. Therefore, bridging the gap between these

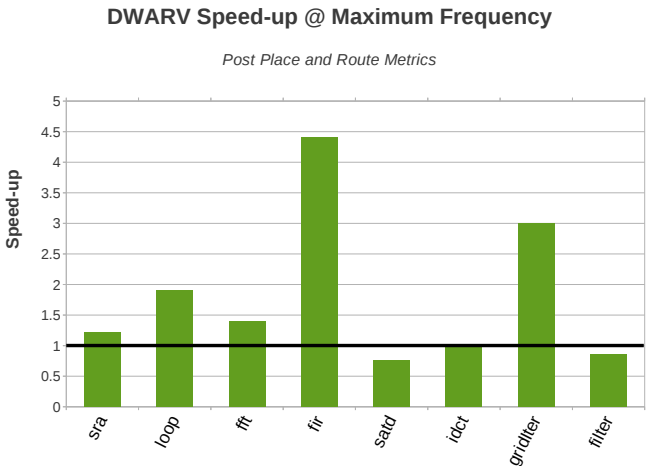

Fig. 2: DWARV 2.0 Speed-ups vs. LegUp times.

interfaces was only a matter of connecting the proper ports to each other, e.g. *DATA_ADDR* to *memory_controller_address*.

DWARV vs. LegUp Comparison

We perform two kinds of comparisons: one that focuses on speed-up and area consumption and one on the restrictions imposed on the C-code. To measure the speed-up and area, we have selected eight kernels for testing. The first four, i.e., *loop, sra, fft, fir*, were extracted from the examples directory in the LegUp distribution, whereas the other four were taken from DWARV 2.0's test bench. All eight functions compiled without any C-language syntax modifications in both tools. Furthermore, the approach described above was followed. The results are summarized in Table II, whereas Figure 2 shows DWARV 2.0 speed-up information for all test cases relative to the times obtained for LegUp. The computed speed-ups were obtained by considering the number of cycles at the maximum frequency reported by the Xilinx post place and route synthesis, except for the *idct* kernel. For this kernel, the initial maximum frequency estimation was used. LegUp *idct* kernel could not be synthesized targeting Xilinx because it contained an instantiation of an Altera proprietary/specific IP block used for integer division. We compared the execution times at kernel level only which gives an indication of the quality of the generated HDL.

Analysing the last column in Table II, we observe that performance wise, DWARV 2.0 gave a speed-up for four kernels, *icdt* provided no improvement or degradation (6th column), whereas the other three functions incurred a decrease in performance. These speed-up numbers were computed by first calculating the **E**xecution **T**ime achieved **A**t **M**aximum **F**requency (ETAMF) reported for the two hardware versions, i.e., $ETAMF = Cycles/Max.Freq.$. Next, $Speedup_{dwarv} = ETAMF_{legup}/ETAMF_{dwarv}$.

With respect to the hardware area, DWARV 2.0 produces less than optimal hardware designs because no optimization passes that target area reduction were used. Our primary focus was functional correctness and to obtain a basis for comparison with future research. As an example of such future research, consider the loop case study. Only by integrating the standard

TABLE II: Evaluation Numbers - DWARV 2.0 vs. LegUp.

Kernel	Slices	Cycles	Max. Freq (xst[2])	ETAMF[1] (xst)	Speedup (xst)	Max. Freq (real[3])	ETAMF (real)	Speedup (real)
sra-legup	370	70	261	0.27	0.82	202	0.35	0.82
sra-dwarv	338	64	290	0.22	1.22	225	0.28	**1.22**
loop-legup	122	292	352	0.83	1.24	251	1.16	**1.30**
loop-dwarv	122	380	368	1.03	0.80	252	1.51	0.77
fft-legup	1980	7377	125	59.02	0.76	98	75.28	0.71
fft-dwarv	3198	8053	180	44.74	1.32	150	53.69	**1.40**
fir-legup	320	223	124	1.80	0.33	80	2.79	0.23
fir-dwarv	1063	127	213	0.60	3.02	201	0.63	**4.41**
satd-legup	1189	132	175	0.75	1.29	150	0.88	**1.31**
satd-dwarv	1201	265	272	0.97	0.77	230	1.15	0.76
idct-legup	N/A	24004	88	320.05	*1.00*	N/A	N/A	N/A
idct-dwarv	9519	41338	151	273.76	*1.00*	75	551.17	N/A
gridIterate_fixed-legup	455	471348	102	4621.06	0.26	100	4713.48	0.33
gridIterate_fixed-dwarv	1343	355810	294	1210.24	3.82	226	1574.38	**2.99**
filter_subband_fixed-legup	342	21464	158	135.85	1.46	103	208.39	**1.17**
filter_subband_fixed-dwarv	386	55137	278	198.33	0.68	226	243.97	0.85

[1]**E**xecution **T**ime **A**t **M**aximum **F**requency
[2]Estimated Maximum Frequency after Behavioural Synthesis
[3]Real Maximum Frequency after Post Place and Route Synthesis

CoSy framework engines *loopanalysis* and *loopunroll*, which annotate respectively unroll simple loops, we decreased the number of cycles for this kernel from 380 to 113. Given the new obtained frequency of 256 MHz, we were able to obtain a speed-up of 1.90 for this example as well (initial numbers are given in Table II were we can see that the first implementation in DWARV 2.0 gave a 0.77 slowdown). Figure 2 shows the final results obtained after this simple optimization was applied. Even though the *loopunroll* engine can provide considerable performance benefits, determining an unroll factor is not a trivial problem. If the unroll factor is too big, the generated VHDL will not synthesize due to lack of available area. Future research will address this problem.

The second comparison we made focused on the extent that the compilers are capable of compiling a large subset of the C-language without requiring substantial rewrites. To do this, we used our internal benchmark, which is a database of 324 kernels from a wide variety of application domains. For example, the cryptography domain contains 80 kernels and the mathematical domain contains 70 kernels. Other domains available in the benchmark are physics, multimedia, DSP, data processing and compression. Simply invoking the two compilers with this database, we observed that DWARV 2.0 is able to generate synthesizable VHDL for 82.1% of the kernels, whereas LegUp for 65.7%. However, LegUp does not support FP operations and, as such, the ability to correctly generate VHDL for our kernel library is degraded. When we ignored FP operations, the performance increased to 87.7%.

V. CONCLUSION

In this paper, we presented the DWARV 2.0 compiler and we did a performance comparison with another academic hardware compiler. We conclude that the current version provides a good basis for future research of hardware related optimizations. One of the most important advantage of DWARV 2.0, compared to the previous version, is that it is highly extensible. Extending the compiler can be achieved by including standard CoSy or custom (new) engines, or can involve extensions in the IR. CoSy's mechanism of extending the IR guarantees that the correctness of the code already written is not affected.

ACKNOWLEDGEMENT

This research is partially supported by the Artemisia iFEST project (grant 100203), the Artemisia SMECY project (grant 100230), and the FP7 Reflect project (grant 248976).

REFERENCES

[1] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov and E. M. Panainte. *The molen polymorphic processor*. In IEEE Transactions on Computers(November 2004). pages: 1363-1375.

[2] Y.D. Yankova, G.K. Kuzmanov, K.L.M. Bertels, G.N. Gaydadjiev, J. Lu and S. Vassiliadis. *DWARV: Delft Workbench Automated Reconfigurable VHDL Generator*, Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL '07): 697-701.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown and T. Czajkowski. *LegUp: high-level synthesis for FPGA-based processor/accelerator systems*. Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '11): 33–36.

[4] Z. Guo, B. Buyukkurt, W. Najjar and K. Vissers. *Optimized Generation of Data-Path from C Codes for FPGAs*, Proceedings of the conference on Design, Automation and Test in Europe - Volume 1 (DATE '05): 112–117.

[5] S. Gupta, N. Dutt, R. Gupta and A. Nicolau. *Spark: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations*. Proceedings of the 16th International Conference on VLSI Design (VLSI '03): 461-466.

[6] Catapult C Synthesis Overview. [Online]. Available: http://www.mentor.com/esl/catapult/overview

[7] C-to-Silicon Compiler. [Online]. Available: http://www.cadence.com/Community/tags/CTOS/default.aspx

[8] Associated Compiler Experts ACE: *CoSy compiler platform*. [Online]. Available: www.ace.nl

[9] The LLVM Compiler Infrastructure Project, 2011. [Online]. Available: http://www.llvm.org.

[10] Univ. of Cambridge. *The Tiger MIPS processor 2010*. [Online]. Available: http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html

[11] Altium Designer 10. [Online]. Available: http://www.altium.com/

[12] Y Explorations (XYI), San Jose, CA. *eXCite C to RTL Behavioral Synthesis 4.1(a), 2010*