2014 International Conference on Embedded and Ubiquitous Computing

High-Level Synthesis in the Delft Workbench Hardware/Software Co-design Tool-Chain

Răzvan Nane*, Vlad-Mihai Sima*, Cuong Pham-Quoc*, Fernando Gonçalves†, Koen Bertels*
* Computer Engineering Lab, Faculty of EEMCS
Delft University of Technology, The Netherlands
Email: ⟨r.nane, p.phamquoccuong, v.m.sima, k.l.m.bertels⟩@tudelft.nl
†Coreworks S.A., Lisbon, Portugal
Email: fernando.goncalves@coreworks-sa.com

***Abstract*—High-Level Synthesis (HLS) is an automated design process that deals with the generation of behavioral hardware descriptions from high-level algorithmic specifications. The main benefit of this approach is that ever-increasing system-on-chip (SoC) design complexity and ever-shorter time-to-market can still be both manageable and achievable. This advantage, coupled with the increasing number of available heterogeneous platforms that loosely couple general-purpose processors with Field-Programmable Gate Array (FPGA)-based co-processors, led to an increasing attention for HLS tool development and optimization from both the academia as well as the industry. However, in order for HLS to fully reach its potential, it is imperative to look simultaneously at local HLS optimizations as well as to HLS system-level integration and design space exploration issues. In this paper, we present the Delft Workbench tool-chain that takes C-code as input and generates, in a semi-automatic way, a complete system. Subsequently, we describe the design and output code optimization of the DWARV 3.0 HLS compiler using the CoSy compiler framework. Based on this experience, we provide an overview of similarities and differences in leveraging this commercial compiler framework to build a hardware compiler as opposed to building a software compiler. Finally, we report speedups up to 3.72x at application level and development times measurable in hours rather than weeks.**

***Index Terms*—Delft Workbench; DWARV; High-level synthesis; Hardware/Software Co-design**

I. Introduction

Reconfigurable computing has gained increasing attention from the industry over the last couple of years as it constitutes a very interesting marriage between the performance of hardware and the flexibility of software. Reconfigurable fabrics such as FPGAs can be used as stand-alone processing units or in combination with a General-Purpose Processor (GPP). In this article, we focus on the latter use, where the reconfigurable device will contain application specific logic, which previously had to be implemented in either dedicated hardware or only in software. Depending on the architecture and usage scenario, the computations executed on the reconfigurable fabric can be changed (at runtime or at compile time) in function of the application at hand. However, for this technology to be adopted on a large scale, important gaps have to be bridged.

One of the challenges is the need for a machine organization that provides a generic way in which different components such as a GPP and various reconfigurable devices can be combined in a transparent way. Another challenge is the need for necessary tools to transform (existing or new) applications in such a way that we can efficiently use the reconfigurable computing units. Such tools are more complex as the application development in this context is no longer a pure software writing effort but assumes substantial hardware design capabilities. For application developers to use this technology, (semi-) automatic support in designing the hardware and to generate the required executable code should be available. In this article, we present the current state of the Delft Workbench (DWB) project that contains the required toolset and which is based on the Molen Programming Paradigm.

The contributions of the paper are the following:

- The presentation of DWB, an operational tool-chain allowing the use of FPGA based accelerators.
- Disscusion of the experience in using the commercial CoSy retargetable compiler framework to implement the Delft Workbench Automated Reconfigurable VHDL generator (DWARV) compiler, which performs HLS.
- Initial results for various heterogeneous platforms.

The remainder of the paper is structured as follows. We first introduce the Molen programming paradigm, which assumes a particular machine organization that combines a GPP with one or more CCU. We then introduce the different steps of the application design process starting at the profiling phase and then moving to transforming the application. Compilation and hardware generation are the final steps after which the applications can be executed. Due to space limitations, we subsequently give a detailed description only for HLS process. We conclude the paper by presenting various results on different heterogeneous platforms.

II. Previous Work

Multiple reconfigurable architectures and processors have been proposed in the last 15 years, both academic and commercial. For a recent overview of these architectures and processors, the reader is referred to [1]. Although the reconfigurable paradigm is becoming mainstream, leading to an increase in available reconfigurable platforms such as Convey HC series, Xilinx Zynq boards or IBM Power-8 servers, automatic end-to-end compilation support for these systems is still not widely available. Tool support is imperative for automatic HW/SW co-design to succeed. In this paper, we briefly present the

978-0-7695-5249-1/14 $31.00 © 2014 IEEE
DOI 10.1109/EUC.2014.28

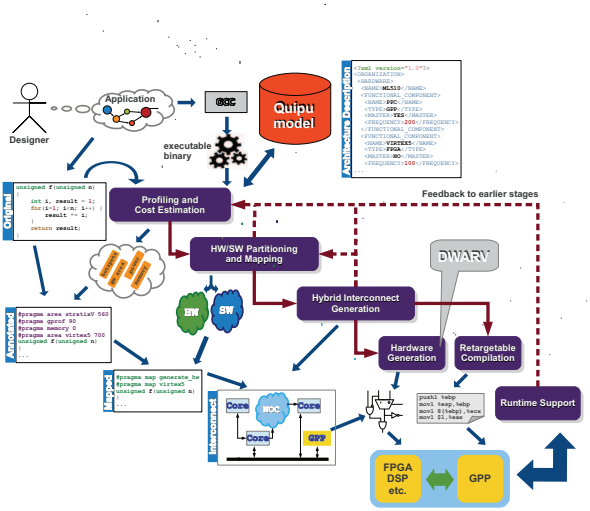


Fig. 1: The Delft Workbench Tool-Chain.

DWB tool-chain that is targeting automatic generation of the complete HW/SW system. This tool-chain includes several stand-alone tools that will be presented in the corresponding section. Each of the tools has its own related work that should be inspected by reviewing its own literature background. In this section, we briefly review only the HLS previous work.

There have been more than 30 HLS tools proposed over the years. Some of the early tools required users to annotate the code extensively before compilation. Furthermore, because the tools were designed with a particular application domain in mind, the input language acccepted was limited to a small subset of C-based languages. As an example we give the ROCCC [2] compiler designed for streaming applications. Vivado HLS [3] and Legup [4] are recent HLS examples that do not substantially restrict the input C-based language. A comprehensive list can be found in [5]. The DWARV [6] compiler integrated in the DWB differs from these compilers mainly because it is built on top of a commercial compiler framework, CoSy [7], which allows the compiler to be extended and configured without major effort.

III. The Molen Programming Paradigm

The Molen programming paradigm is a sequential consistency paradigm in which an application can be partitioned in such a way that certain parts can run (in parallel) on the reconfigurable fabric and other parts run on the GPP. This paradigm assumes the Molen machine organisation [8], which defines the interaction between a GPP and the Custom Computing Unit (CCU)s, implemented on the FPGA. It consists of a one time instruction set extension that allows the implementation of an arbitrary number of CCUs. The four basic instructions needed are SET, EXECUTE, MOVTX and MOVFX. By implementing the first two instructions (SET/EXECUTE), a hardware implementation can be loaded and executed in the reconfigurable processor. The MOVTX and MOVFX instructions are needed to provide the communication between the reconfigurable hardware and the GPP.

IV. Components of the Delft Workbench

In this section, we provide a high-level view of the main steps that are needed to port an application to a heterogeneous platform. We will highlight for each of these steps the tools from DWB toolchain (Figure 1) that provide support to the user in order to efficiently implement a heterogeneous system.

A. Profiling and Cost Estimation

Profiling consists of characterizing the application with respect to criteria such as execution time, communication, or resource consumption with the purpose to drive subsequent partitioning and mapping of an application on heterogeneous platforms. The main purpose of collecting this information is to support the later mapping and partitioning step. For this purpose, we characterize an application by determining the computational hotspots, the communication bottlenecks, and the resource-intensive parts within an application.

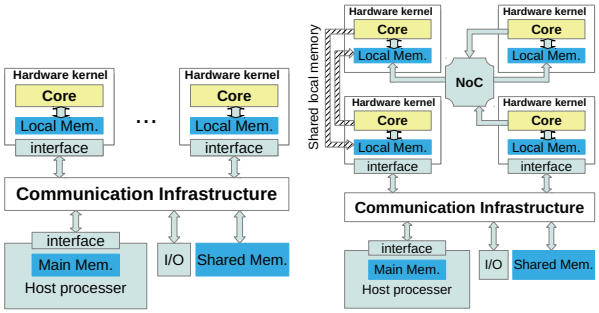
The Quipu [9] tool, based on advanced statistical methods, predicts the hardware resources needed to implement certain software parts in hardware. As a result, the approach is not restricted to one platform or toolchain but allows the generated models to be re-calibrated for various other tools or architectures, in contrast to most existing techniques. As Quipu models are linear (or polynomial) equations, it is possible to make predictions very fast, which allows for rapid Design Space Exploration (DSE). On the tools side, the approach is represented by tools that generate the initial models and allow a developer to obtain the information for the developed applications.

Quad [10] consists of several tools developed to provide a comprehensive overview of the memory access behavior of an application, as well as, to extract fine-grained detailed memory access related statistics. The Quad core tool primarily detects the actual data dependencies at the function-level. These dependencies provide valuable insight into data flows in an application and help guide mapping schemes, such as those found in Section IV-B, to reduce communication latency or identify parallelism.

Quad measures data dependency as producer/consumer bindings. More precisely, actual data dependency arises when a function consumes data that was produced earlier by another function. Specifically, the following profile data is extracted by the Quad core tool:

- the exact amount of data transfer (in bytes) between each pair of producer/consumer functions,
- the exact number of *Unique Memory Addresses* (UnMAs) used in the transfer,
- the exact number of *Unique Data Values* (UnDVs) that is read by the consumer.

Based on the efficient *Memory Access Tracing* (MAT) module implemented in the Quad core tool, which tracks every single access (read/write) to a memory location, a variety of statistics related to the memory access behavior of an application can also be measured. This includes, for example, the proportion of local to global memory accesses in a particular



(a) (b)

Fig. 2: (a) The generic HW accelerator architecture; (b) The generic HW accelerator system with our custom interconnect for the communication of kernels

function call. All Quad related tools are implemented utilizing the Pin [11] Dynamic Binary Instrumentation (DBI) platform.

B. Partitioning and Mapping

In this part, we present a technique that allows to decompose the application in parts that can be mapped to the target architecture. The technique is primarily focused on making partitioning decisions based on data communication provided by the profilling framework.

Traditionally, the primary objective of hardware/software partitioning in heterogeneous reconfigurable systems is to improve the execution performance, hence, to gain speedup. In order to achieve this goal, the program parts with higher execution time contributions are usually mapped to the hardware, while the parts with lower contributions are executed on the general-purpose processor. Nevertheless, it has turned out that the key hurdle to limit the speedup is the well-known *memory bottleneck* problem. The problem is even intensified in case the potential hardware kernel candidates are heavily communicating with the kernels in the software part.

To address this problem, we utilize the profile information provided by the Quad toolset to minimize, as much as possible, the data communication between the hardware and the software boundary. This is mainly achieved by *merging* tightly coupled kernels, respecting the Molen machine restrictions, and kernel *splitting* in case the hardware area limitation prevails.

C. Interconnect Generation

An important challenge in a hardware accelerator system is to get the data to the computing core that needs it. The goal is of course to hide the communication delay such that a performance improvement can be observed. The resource allocation decision requires detailed and accurate information on the amount of data that is needed as input and what will be produced as output. Evidently, there exists dependencies between computations and data produced by one kernel will be needed by another kernel. In order to have an efficient allocation scheme where the communication delays can be hidden as much as possible, a detailed profile of the data communication patterns is necessary for which then the most appropriate interconnect infrastructure can be generated.

Using the detailed profile of the data communication patterns, a kernel knows exactly which kernels will consume its output. Therefore, the kernel can deliver its output directly to the consuming kernels when the output is available. To support this model in a conventional hardware accelerator system, beside the existing communication infrastructure, a custom interconnect for the kernels is implemented. The custom interconnect in our work includes a Network on Chip (NoC), shared local memory mechanism or both. We share the local memories of two kernels that only communicate with each other, while the NoC is used for a group of more than two communicating kernels. Figure 2(b) depicts our concept system in which a custom interconnect is implemented to improve data communication of kernels in a generic hardware accelerator system (Figure 2(a)). More detailed information can be found in [12] and [13].

D. System Level Optimizations

Another important step in porting an application to a heterogenous architecture is system level optimization. One example of such optimization is the scheduling of SET and EXECUTE instructions and the decision on how to allocate the available FPGA area, denoted as spatio-temporal compilation [14]. This optimization needs to be implemented in the GPP compiler as the GPP is the one who controls the execution of the FPGA.

The ‘temporal’ refers to the point in the execution flow at which the configuration of the CCU is started. Given the latency the configuration operation incurs, the compiler will schedule the SET operation such that the configuration time is as much as possible hidden. The ‘spatial’ aspect refers to the available area on the FPGA. Each CCU occupies a certain amount of the area and again it is the compiler’s task to determine how many CCUs will fit at any given time on the FPGA. To address these spatio-temporal constraints, we introduce advanced instruction scheduling and area allocation algorithms. These algorithms minimize the number of executed hardware reconfiguration instructions based on the application’s specific features while taking into account the available area.

E. Hardware Generation

While the first fork in Figure 1 concerned the (software-side) compiler scheduling of the SET and EXECUTE instruction, the second fork involves the hardware generation for the kernels that have been selected for hardware acceleration by the previous profile and partitioning steps. In the DWB, this is done with the help of the DWARV hardware compiler, which is described in the next section. The output of DWARV is synthesizable VHDL. The toolchain will automatically invoke the architecture specific tools that generate bitstreams from the generated VHDL.

F. Final linking

In the final stage, the software executable and all the generated bitstreams are linked together into one executable, ready to be run on the platform. This final executable will include also functions that are part of the platform specific Molen Abstraction Layer, providing the necessary runtime support.

V. High-Level Synthesis using CoSy

In this section, we describe the DWARV HLS compiler. Our focus is on describing the specific features a hardware compiler has when compared to a software compiler and comment on their implementation in DWARV.

A. CoSy Compiler Framework

The CoSy [7] compiler development system is a commercial compiler platform licensed by ACE Associated Compiler Experts. It is composed of several tools building on innovative concepts that allow the user to easily build and extend a compiler. The central concept in CoSy is the *Engine* that can be regarded as an abstraction of a particular transformation or optimization algorithm. The framework includes a total number of 208 engines that target different parts of the compilation flow, from the frontend (i.e., code parsing and Intermediate Representation (IR) creation) processing, high-level (e.g., loop optimization, algebraic optimizer, code analysis, lowering transformations) and low-level (e.g., register allocation, instruction selection and scheduling) transformations and optimizations to backend code generation template rules used in the final emission step. The general functionality of new, user-defined, engines can be programmed in C or in C++ while for IR specific initializations and assignments, CoSy-C, an extension of the C-language, is used.

B. Software vs. Hardware Compilers

A compiler is composed of several main passes that transforms the input high-level language code to a format accepted by the target machine. We define a software compiler as a compiler that transforms a High-Level Language (HLL) to assembly instructions, whereas a hardware compiler is one that transforms a HLL code to an equivalent Hardware Description Language (HDL) representation.

Building a hardware compiler is not very different than building a software compiler. In principle, the standard steps of compilations (scanning and parsing, IR generations, IR optimizations, instruction selection and scheduling, register allocation) can be applied. However, because on (re)configurable hardware the number of resources is not fixed, and, as such, we can (to some extent) accommodate as many resources as required, we have to change how some of the traditional software compiler passes are applied in the context of hardware generation. Although this kind of analysis can apply to various compiler transformations and optimizations, for the scope of this section, we discuss here only the ones that we considered more relevant: Common Subexpression Elimination (CSE), operation chaining, register and memory space allocations.

CSE is an optimization that computes common expressions once, places the result in a register or memory, and refers to this location for subsequent uses. The advantage is that we can save time and resources by not computing the expression again because loading the already computed value is faster on a Central Processing Unit (CPU). However, for hardware generation this has to be carefully considered. That is, if the allocated register and the expression where this replaced expression should be used is not in the immediate proximity of the calculation, the routing to that place might actually decrease the design frequency. Therefore, applying this optimization is not always useful in HLS. In this case, replacing only a particular subset and recomputing the expression for others to enforce a better locality of the operations, would provide better results.

Operation chaining is an optimization that is specially designed for hardware compilers. The clock period is a design dependent parameter which entails, once specified, specific optimisations. Dependent instructions can be scheduled in the same clock cycle if their cumulative execution time is less than the target cycle time. In software, instructions operate only on registers implying that temporary results or variables need to be allocated to a register. When generating hardware, we have two distinct cases. For temporary results that are used in the same clock cycles, no actual register needs to be allocated. For each result that is needed in a different clock cycle, a new register is allocated. To implement this, the hardware register allocation engine will use the information provided by operation chaining to determine what the type of the result is for each generated operation.

Independent memory spaces is a concept which can ocassionally be found in GPPs but it is essential for hardware. The hardware generated by DWARV can access a number of independent memory ports, which allows parallelizing memory accesses and improving performence. As the support for memory spaces is present in Cosy, the effort to implement a variable number of independent memory spaces that can be accessed in parallel was greatly reduced.

C. IP Library

In order to provide a generic HLS compiler, we need to introduce the concept of library. This is similar to what exists in the software world where there can be multiple implementations for a library depending on the target architecture. An additional complication for hardware is that even basic language operations (addition, multiplication) can be implemented differently depending on architecture, the operating frequency and resources used.

In order to support all these features, we have implemented IP library support that can include both basic operators such as floating point operations, and generic functions. The library targets currently Xilinx FPGA-s, but can be easily adapted to other platforms. An example of the information provided in the library descriptor is given in Table I. This information is used by DWARV at compile time and the information is used to generate the appropriate VHDL.

TABLE I: Library description IP examples

Library Field	Description	Oper. example	Oper. example
IP Name	Component Name	fp_sp_mult	fp_sp_mul_add
Input Port 1	First port name	a	in_0
Input Port 2	Second port name	b	in_1
Input Port 3	Third port name	N/A	in_2
Output Port	Output port name	result	result
OP Type	Operation type	MUL	MULADD32
OP Size	Operands size	FP32	FP32
Delay	Latency of IP	6	13
Frequency	Max. frequency	220	220

VI. Experimental Results

To validate and asses the Delft Workbench toolchain, algorithms taken from various domains have been implemented for which significant speedups have been achieved. In the following, we will not elaborate on all the stages of the design process, but rather emphasize the toolchain's HLS capabilities by reporting execution times at both kernel- and application-level as well as showing the energy-efficiency improvements. The performed experiments have been conducted on two different platforms designed for different application domains, i.e. embedded and high-performance computing domains. We will also show the kernel results for the latest version of DWARV compiler.

A. Application-level Results for High-Performance Domain

As computing platform for the high-performance domain we used the Convey HC-2^{ex} [15]. The machine includes two Intel Xeon X5670 processors and four Virtex 6 xc6vlx760 FPGA devices. The host processors and the accelerated kernels, located on the FPGA, can communicate through a DDR shared memory.

The Canny edge detection application [16] (ANSI C version provided by University of South Florida) is used as case study. Using profile information, we determined the two most time consuming functions (Guassian and Non_Max_Supp) and it was decided for them that acceleration on FPGA is beneficial. Using the DWARV compiler, we generated the VHDL kernel implementations and using the Xilinx tools we routed and placed them on the FPGA. Each function has eight computing kernels in each FPGA device, i.e. 32 kernels for each function. The kernels run at 150MHz while the host processor works at 2.93GHz. The input image used in the test has the size of 1024x1024 pixels and is divided into 32 segments.

In order to assess the performance improvement, we first run the Canny application on the host processor. The software is compiled by GCC using -O2 optimization level. We then run the accelerated application in which the two aforementioned functions are executed by the generated hardware while the other functions are processed by the host processor. Table II shows the execution time and speed-up of the kernels and the overall application. As shown in the table, we achieved a speed-up of the kernels and of the overall application by up to 1.27× and 1.13× when compared to the host processor.

The hardware resources used are shown in Table III.

TABLE II: Execution time and speed-up for Canny application on Convey HC-2^{ex}

	Software time	Accelerator time	Speed-up
Gaussian	64.878ms	56.453ms	1.15×
Non_Max_Supp	20.261ms	15.856ms	1.27×
Application	119.11ms	106.28ms	1.13×

TABLE III: Hardware resource usage for the kernels, for the Virtex 6 xc6vlx760 FPGA device

Resource type	Amount	Percentage
LUT	187,581	39%
Register	197,374	20%
DSP	344	39%
BRAM	584	81%

B. Application-level Results for Embedded System Domain

The second use case targets the embedded domain. The applications are the MPEG and G729 audio encoder executed on a Xilinx ML510 board. On this FPGA we implement a system on chip, using the embedded PowerPC processor running at 400MHz and several other components (Memory Controllers, UART, Timer, Interrupt, etc.).

The execution time of the encoders depend on the configuration of the encoders. For example, in the MPEG encoder, two important parameters that affect the overall performance are the encoding layer (**L1** or **L2**) and the psychoacoustic model (simple model (**P1**) or complex model (**P2**)). For the G729, the parameters that define the characteristics of the encoded data are the output bitrate (6.4, 8 or 11.4 kbps) and the DTX mode (enabled or disabled). For each application, all possible combinations were tested: four and six combinations for the MPEG respectively G729. Based on output of the profiling and partitioning data obtained from the first part of DWB, three respectively five functions were selected to be implemented as CCUs generated by DWARV.

The overall execution times are presented in Table IV, that is, the execution times for the complete encoders (MPEG or G729). The application was exercised with all possible combinations of the input parameters.

TABLE IV: Overall Execution Times on the ML510 Platform

App.	Operation Mode	Software (seconds)	DWARV (seconds)	Speedup (%)
MPEG	L1-P1	6.25	4.67	25.2
	L1-P2	6.13	3.77	38.5
	L2-P1	4.18	2.72	34.8
	L2-P2	4.65	2.74	41.0
G729	6.4kbps-noDTX	11.87	9.30	21.7
	6.4kbps-DTX	8.95	7.19	19.7
	8kbps-noDTX	13.55	8.96	33.9
	8kbps-DTX	9.99	6.97	30.2
	11.8kbps-noDTX	17.07	16.14	5.4
	11.8kbps-DTX	13.84	14.87	-7.4
	TOTAL	**96.48**	**77.13**	**19.8**

C. Application-level Results for Embedded System Domain using Interconect Generation

To test the interconnect generation, we use again the Xilinx ML510 board.

In this case study, we accelerated the applications using only standard communication infrastructure (Xilinx PLB bus) as well as our own generated NoC.

Four applications are used: the Canny edge detection application [16], JPEG decoder application [17], KLT feature tracker [18] and Fluid simulation [19]. The hardware kernels for each application are generated using the DWARV compiler.

The first step is to run the applications on the GPP processor alone. We then run the applications using only standard communication infrastructure and get computation time and communication time of kernels and overall application. Figure 3 shows the speed-up of the baseline system when compared to the SW as well as ratio between communication time and computation time. The baseline system achieves speed-ups of up to 4.23× for the kernels and 2.93× for the overall application. However, the performance of the baseline system is slower than the SW in case of the *jpeg* application because of the large communication time. Although the baseline system achieves a speed-up of 1.62× for the overall application and of 1.98× for the kernels compared to the SW in average, the communication time of the kernels is larger than the computation time (the ratio is about 2.09×). Therefore, reducing the data communication time can lead to a significant improvement in system performance.

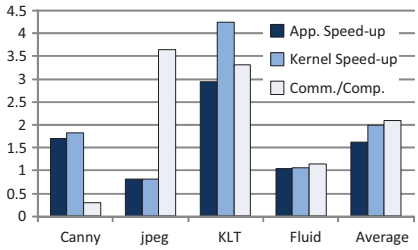


Fig. 3: The speed-up of the baseline system compared to the software only execution and the ratio between communication and computation time.

The second step is to perform an experiment with the custom interconnect generated in the context of the Delft Workbench toolchain. To this purpose, we developed a 2×2 crossbar for the shared local memory solution and adapted the NoC presented in [20] into our system. Table V shows the hardware resource usage and maximum frequency for hardware modules used as interconnect. The crossbar does not introduce any communication overhead because data does not need to be encoded and decoded when transferred through the crossbar.

The proposed system is implemented at the same frequency of the baseline system. Here, we present the details of the *jpeg* decoder from the PowerStone benchmark. The four functions identified as most computationally intensive and suitable for hardware generation are *huff_dc_dec*, *huff_ac_dec*, *dquantz_lum* and *j_rev_dct*. The most computationally intensive function *huff_ac_dec* is duplicated to improve overall execution time. The rest of the functions in the application

TABLE V: Hardware characteristics of interconnect components

Component	**Resource** (LUTs/Registers)	**Max. frequency**
Bus	1048/188	345.8MHz
Crossbar	201/200	N/A
NoC Router	309/353	150MHz
NA[1] HW Accelerator	396/426	422.5MHz
NA[1] local memory	60/114	874.2MHz

[1]: Network adapter for the communication between the HW module and the NoC

are executed on the GPP processor, which is a PowerPC in this experiment. The application is then analyzed in order to extract the data communication graph using the QUAD tool. The results of this analysis are depicted in Figure 4.

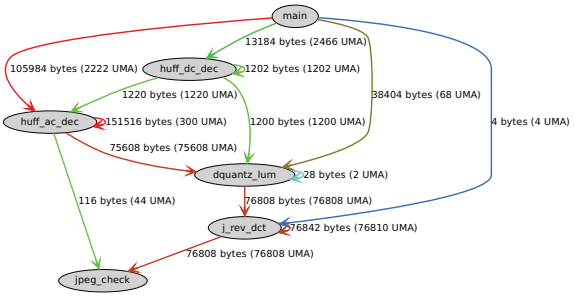


Fig. 4: Data communication profiling for the *jpeg* decoder

The graph contains relevant information with respect to the data communication patterns in the application. Function *dquantz_lum* sends data only to *j_rev_dct* and function *j_rev_dct* consumes data generated by the host processor and *dquantz_lum*. Based on this, we decide to apply the shared local memory solution to these two functions.

The NoC is used to connect of *huff_dc_dec*, two kernels of *huff_ac_dec* and *dquantz_lum*. As shown in the graph, *huff_dc_dec* only consumes data coming from the host processor and ony sends data to kernels *huff_ac_dec* and *dquantz_lum*.The local memory of *huff_dc_dec* is connected to the system communication infrastructure (the PLB bus in this case). The interconnect for all other kernels is deduced in a similar way, resulting in the overall architecture shown on Figure 5.

As stated above, we performed our experiment with the BRAM as the local memories of the kernels. Each BRAM has two ports while local memories of two *huff_ac_dec* kernels (*kernel* 1 and *kernel* 2 in Figure 5) are accessed by three different components (the host, the NoC adapter and the kernel core). In those cases a multiplexer is used.

Table VI and Figure 6 show the speed-up of the overall application (column 2 and 4) and of the kernels (column 3 and 5) of the proposed system with respect to both software and the baseline system. As shown in the table, the optimized applications achieve a speed-up of up to 3.72× when compared to software.

We also assess the hardware resources used in the three

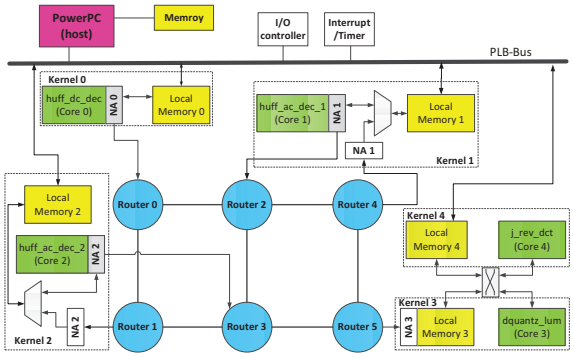


Fig. 5: The proposed system architecture for the *jpeg* decoder application

TABLE VI: Speed-up of the optimized system with respect to software and the baseline system

App.	w.r.t Software		w.r.t Baseline	
	App.	Kernels	App.	Kernels
Canny	3.15×	3.88×	1.83×	2.12×
jpeg	2.33×	2.5×	2.87×	3.08×
KLT	3.72×	6.58×	1.26×	1.55×
Fluid	1.66×	1.68×	1.59×	1.60×

scenarios. The results are given in terms of the number of FPGA look-up tables (LUTs) and the number of FPGA registers in Table VII. The NoC-only system is a system in which only a NoC is used to implement a parallel solution. As shown in the table, our optimized system saves up to 33.1% LUTs and 30.2% registers compared to the NoC-only system. Without the presented strategy, the system is either only bus-based or only NoC-based. The bus-based system (baseline) is a low performance system while the NoC-only system uses more hardware resources than our optimized system.

TABLE VII: Hardware resource utilization (LUTs/Registers)

App.	Baseline	Our system	NoC only	Solution
Canny	9926 12707	15227 18657	17894 21059	NoC, SM[1], P[2]
jpeg	11755 11910	20837 20900	23180 23188	NoC, SM[1], P[2]
KLT	4721 5430	4921 5631	7358 8070	SM[1]
Fluid	19125 28793	24156 36100	24552 36110	NoC

[1] SM: Shared memory; [2] P: Parallel

D. Energy Efficiency Results

Finally, for the embedded system domain we present the energy efficiency results. Table VIII presents the power and energy figures for the implementation in the ML510 FPGA development board of the audio decoding applications. The measurements were performed on the FPGA board using the System Monitor feature provided by Xilinx, while the measurement data was accessed via the JTAG port. The Xilinx ChipScope Pro tool provides access to the System Monitor over the JTAG port.

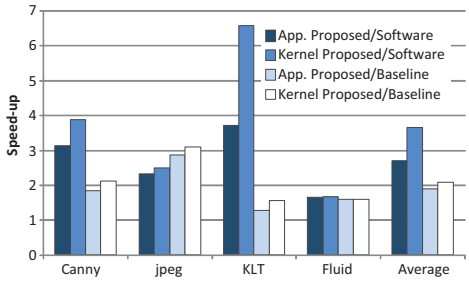


Fig. 6: The overall application and the kernels speed-up of the baseline and our systems

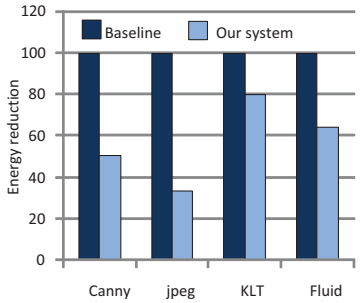


Fig. 7: Energy consumption comparison between the baseline system and our system normalized to the baseline system.

The energy results are shown in the last column of Table VIII. When the hardware/software implementation is used, the energy consumption to encode the MPEG and the G729 audio files using the whole set of combinations is smaller. The reduction in the energy consumption is approximately 15% on average, which shows the effectiveness of the results generated by the DWB tool-chain, and particularly by the DWARV HLS tool.

We performed a similar analysis for the applications in the interconnect generation case. Figure 7 presents the comparison of the energy consumption between the baseline system and our optimized system. We used the Xilinx XPower Analyzer 13.2 tool to estimate the power consumption of each application in the two systems. The energy consumption is given by the product of the power consumption and the execution time. The power consumption is similar for both systems, therefore, taking into account the reduction in execution time, our optimized system consumes less energy. As shown in the figure, our system outperforms the baseline in all applications. The maximum energy saved is 66.5% for the *jpeg* application.

E. Kernel-level Results for Next-Generation of DWARV

All the results presented so far are generated with DWARV 2.0. In this section and shown in Table IX, we present the results with the next version of DWARV, namely DWARV 3.0. Compared to the previos version, the speedup is between 1.27x and 3.52x. This is due to a better operator chaining algorithm and to the fact that the new version uses as much as possible independent memory spaces for both input and output data.

TABLE IX: Performance and Area Metrics for DWARV 2.0 and DWARV 3.0 .

Function Name	Benchmark	DWARV 2.0 - xc5vfx130t-2-ff1738					DWARV 3.0 - xc7k325t-2-ffg900					Speedup
		Cycles	FMax	Regs	LUT	Exec. Time (us)	Cycles	FMax	Regs	LUT	Exec Time (us)	
adpcm-encode	CHSTONE	1217	197	7621	5933	6.18	509	227	2766	3452	2.24	2.75
aes-encrypt	CHSTONE	14151	173	31412	14619	81.80	7948	191	17582	10039	41.61	1.96
aes-decrypt	CHSTONE	26278	179	30222	13274	146.80	7951	191	15871	8199	41.63	3.52
gsm	CHSTONE	18778	175	14588	12230	107.30	9857	235	3728	6419	41.94	2.55
sha	CHSTONE	ERR	ERR	ERR	ERR	ERR	212499	229	13722	33549	927.94	-
matrixmult	DWARV	678	211	651	475	3.21	420	307	391	291	1.37	2.34
cmultconj	DWARV	71	406	785	491	0.17	37	302	999	880	0.12	1.41
satd	DWARV	243	279	3295	3105	0.87	84	124	713	1762	0.68	1.27
sobel	DWARV	25055	263	1394	1186	95.27	13428	286	711	796	46.95	2.02
viterbi	DWARV	ERR	ERR	ERR	ERR	ERR	33782	280	5782	8146	120.65	-
belmannford	DWARV	11098	273	1224	1035	40.65	6701	368	716	623	18.21	2.23
{**AVERAGE ; GEOMEAN**}												{**2.24;2.14**}

TABLE VIII: Power and Energy Results

	Average Power (Watts)	Execution Time (seconds)	Energy (Joules)
Idle	2.340	–	–
Software	2.368	96.48	228.46
Hardware/Software	2.500	77.33	193.32

We notice here two cases, i.e. *sha* and *viterbi* functions, that DWARV was able to compile, however, the simulation results did not match the software reference and, as a result, and we did not provide results for them.

VII. Conclusion and Future Work

In this article, we have sketched the different steps involved in developing applications that will run on a heterogeneous platform such as the Molen polymorphic processor. We have also given a detailed description of the HLS compiler needed in order to generate code for reconfigurable architectures. Specific about designing applications for such platforms is the blend of hardware and software development. As both require different techniques and skills, tools are required to provide sufficient support to fill in the gaps as much as possible. We have pointed out the different steps required to develop a new or modify an existing application that can be executed on such platforms.

Through experimental implementation, we have shown the DelftWorkbench toolchain can provide both significant results in performance (up to 3.72x at application level) and in energy savings (up to 15% measured savings), while not making the development process excedingly complex.

References

[1] A. Chattopadhyay, "Ingredients of adaptability: A survey of reconfigurable processors," *VLSI Design*, Jan. 2013. [Online]. Availabl1e: http://dx.doi.org/10.1155/2013/683615

[2] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, May 2010, pp. 127–134.

[3] "Xilinx, inc." [Online]. Availabl1e: http://www.xilinx.com

[4] 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, pp. 33–36. [Online]. Availabl1e: http://doi.acm.org/10.1145/1950413.1950423

[5] R. Nane, "Automatic hardware generation for reconfigurable architectures," Ph.D. dissertation, 2014.

[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 *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, Aug 2012, pp. 619–622.

[7] Cosy compiler Framework. [Online]. Availabl1e: http://www.ace.nl/compiler/cosy

[8] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The molen polymorphic processor," *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1363–1375, 2004.

[9] R. Meeuws, C. Galuzzi, and K. Bertels, "High level quantitative hardware prediction modeling using statistical methods," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, July 2011, pp. 140–149.

[10] S. A. Ostadzadeh, R. Meeuws, C. Galuzzi, and K. Bertels, "Quad - a memory access pattern analyser," in *ARC*, 2010, pp. 269–281.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Availabl1e: http://doi.acm.org/10.1145/1065010.1065034

[12] C. Pham-Quoc, J. Heisswolf, S. Wenner, Z. Al-Ars, J. Becker, and K. Bertels, "Hybrid interconnect design for heterogeneous hardware accelerators," in *Proc. Design, Automation & Test in Europe Conference & Exhibition*, Grenoble, France, March 2013.

[13] C. Pham-Quoc, Z. Al-Ars, and K. Bertels, "Automated hybrid interconnect design for fpga-based accelerators using data communication profiling," in *Proc. 28th IEEE International Parallel & Distributed Processing Symposium Workshop*, Phoenix (Arizona), USA, May 2014.

[14] E. M. Panainte, K. Bertels, and S. Vassiliadis, "Instruction scheduling for dynamic hardware configurations," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 100–105.

[15] C. Computer. [Online]. Availabl1e: http://www.conveycomputer.com/

[16] J. Canny, "A computational approach to edge detection," *Pattern Analysis and Machine Intelligence*, pp. 679 –698, 1986.

[17] J. Scott, L. H. Lee, J. Arends, and B. Moyer, "Designing the low-power m*core architecture," 1998.

[18] J. Shi and C. Tomasi, "Good Features to Track," in *Computer Vision and Pattern Recognition*, 1994.

[19] J. Stam, "Real-time fluid dynamics for games," in *the Game Developer Conference*, 2003.

[20] J. Heisswolf, R. Koenig, and J. Becker, "A scalable NoC router design providing QoS support using weighted round robin scheduling," in *ISPAW*, 2012, pp. 625– 632.