Abstract—Many of today's embedded multiprocessor systems are implemented as heterogeneous systems, consisting of hardware and software components. To automate the composition and integration of multiprocessor systems by making tools interoperable, the IP-XACT standard was defined to describe hardware IP blocks and (sub)systems. However, the IP-XACT standard does not provide sufficient means to express Reconfigurable Computing (RC) specific information, such as Hardware dependent Software (HdS) meta-data, which prevents automated integration. In this paper, we propose several extensions such that the HdS can be generated automatically. To this purpose, we investigate the capabilities of the IP-XACT standard to model tool requirements for tool interoperability and automated integration of HdS. Based on the proposed integration framework, we elaborate on the expressiveness of IP-XACT to model HdS and comment on its shortcomings. Furthermore, we validate one specific extension and demonstrate the interoperability of the approach based on an H.264 decoder application case study. For this case study we achieved an overall 30.4% application-wise speed-up and we reduced the development time of HdS from days to a few seconds.

I. INTRODUCTION

A widely adopted practice within Reconfigurable Computing (RC) design is to accelerate part(s) of applications, using custom hardware architectures, that are specifically tailored for a particular application. These specialized architectures can be Intellectual Property (IP) blocks written at the Register Transfer Level (RTL) by a designer, or IP blocks generated by a High Level Synthesis tool from a functional specification written in a High Level Language (HLL) [9]. To cope with the diversity of IP blocks coming from different sources, IP-XACT [8] was introduced, which is an IEEE standard based on XML schemas. Using IP-XACT, hardware components can be described in a standardized way. This enables automated configuration and integration of IP blocks, aiding hardware reuse and facilitating tool interoperability [4].

In a Hardware/Software (HW/SW) system, connecting the different HW and SW components, using for instance buses or point-to-point connections, is not sufficient to fully implement a system. Typically, a SW component connected to a HW component needs a driver program to control the HW component. The driver associated with a certain HW component is also known as Hardware dependent Software (HdS) [1]. IP blocks that can be controlled from a SW component, are typically shipped with a particular HdS that ensures proper control from SW. However, in RC systems, the IP blocks are automatically generated by HW tool-chains for application kernels selected for hardware acceleration. Therefore, the HdS driving these new hardware blocks has to be generated automatically as well. The compilation process in such RC systems, i.e., from HLL application source code to a combined HW/SW executable, is done by different tools, such as partitioning, mapping and HW/SW generation tools. This implies that there is no central place from where the HdS can be generated. That is, the compiler used to generate the IP has no knowledge about what HW primitives are used for example to communicate data in the system, which prevents it from generating a proper driver. This information is available, however, in the partitioning and mapping tool. Therefore, to solve this problem, we adopt a layered HdS solution that is gradually generated and passed along the tool-flow, until the final integration step is reached. Furthermore, to allow the tools involved in this HdS generation process to communicate seamlessly with each other, we need to describe the software requirements of each step in IP-XACT as well. One example of such a software requirement is the number of function input parameters. However, unlike the RTL constituents of an IP block, which can be already described using the current IP-XACT standard, there is no standardized way to describe the driver information for an IP.

The current IP-XACT standard does not provide sufficient means to describe the steps necessary to compile, assemble and link the different HdS constituents. Therefore, fully automating the integration of both HW and SW components into a final system is currently not feasible. To enable a fully automated HW/SW integration, there is a need to model not only the RTL of IP blocks, which we refer to as level 0 schema descriptions, but also to model the HdS associated with the IP block. We refer to these new HdS related extensions as level 1 schema descriptions. The IP-XACT standard currently addresses level 0 and provides only limited means for level 1 schema descriptions. In this paper, we elaborate on the expressiveness of IP-XACT for describing HdS meta-data. Furthermore, we address the automation of HdS generation in the RC field, where IPs and their HdS are generated on the fly, and therefore, are not fully predefined. The contribution of this paper can be summarized as follows:

- We combine two proven technologies used in MPSoC design, namely IP-XACT and HdS, to automatically integrate different architectural templates used in RC systems.
- We investigate and propose IP-XACT extensions to allow automatic generation of HdS in RC tool-chains.

The rest of the paper is organized as follows. Section II gives a brief overview of IP-XACT with an emphasis on the parts relevant to HdS based integration in our work. Section III presents other HdS solutions and already proposed IP-XACT extensions. Section IV describes a HdS generation case study and investigates the IP-XACT support for automation. Section V elaborates on the identified shortcomings and proposes
IP-XACT extensions to support level 1 schema descriptions. Section VI presents implementation details to support the automated tool-flow. Section VII respectively Section VIII validates the automated integration and concludes the paper.

II. IP-XACT Standard and Hds

The IP-XACT standard (IEEE 1685-2009) [8] describes an XML schema for meta-data modelling of IP blocks and (sub)systems. The meta-data is used in the development, implementation and verification of electronic systems. The IP-XACT standard also describes an Application Programming Interface (API) to provide tool access to this meta-data. Tools compliant with this interface are able to interpret, configure, integrate and manipulate IP blocks that comply with the proposed IP meta-data description. Fig. 1 illustrates a typical IP-XACT Design Environment (DE). This DE consists of three main parts: an IP core library, an IP-XACT compliant design tool and Generators, i.e. programs that can access the IP-XACT meta-data to realize vendor specific configurations or interactions with other tools.

The IP-XACT standard is currently composed of seven top-level schemas, described briefly in the following list:

- **Component** defines a HW IP block.
- **Bus Definition** defines the type attributes of a bus.
- **Design** defines the configuration of and interconnection between components.
- **Generator Chain** defines the grouping and ordering of generators that create, configure and instantiate a component in the final design.
- **Design Configuration** defines additional configuration information for a generator chain or design description.
- **Abstraction Definition** defines the representation attributes of a bus.
- **Abstractor** defines an adaptor between interfaces of two different components.

In this paper, we focus on the Component schema for associating Hds to HW IP blocks and we focus on Generator-Chain schemas to express compiler specific requirements. The current schema provides limited support for software descriptions. Namely, one can only attach software file-sets to a component and describe the function parameter’s high level types. However, it does not offer the possibility to assign semantics to what is inside this attached file-set and how that should be used during integration of a complete system. Furthermore, it lacks means to model chains in which complex software generators are to be integrated. This usually implies that to properly use it, the name of the compiler alone does not suffice. For instance, one needs to describe compiler specific input files as well. These proposed software related extensions, defined in this paper as IP-XACT level 1 schema descriptions, will be detailed further in Section V.

III. RELATED WORK

Several industry and research institutes have indicated the need for a standard methodology to describe hardware blocks and investigated how to model in a common way different IPs generated from various tools. The goal was to facilitate design reuse in Multi-Processor Systems on Chip (MPSoCs). An important contribution was made by the SPIRIT consortium, which was concluded with the IP-XACT specifications being adopted as the IEEE 1685-2009 standard. The standard is currently maintained by Accellera [8]. This enables design reuse by unifying ways to describe IPs with well-defined XML Schemas for meta-data that document its characteristics. This facilitates the automation of the configuration and integration of these IP blocks. The OpenFPGA CoreLib [11] working group focused on examining the IP-XACT Schema and proposed extensions for facilitating core reuse into HLLs. The group has proposed Structural interface extensions to include HLL data types for ports, temporal interface extensions to describe the timing of the core signals, and control interface extensions to specify how the core must be synchronized with the rest of the circuit. Wirthlin et al. [12][13] used XML to describe common IP block elements and defined their own schema using IP-XACT syntax. They proposed a lightweight version intended for and dealing only with information needed in Reconfigurable Computing (RC) systems, e.g. interface specifications and capturing HLLs data types information.

Other IP-XACT related research is focusing on extending the schema to incorporate semantic information about IP elements. Kruijtzer et al. [5], proposed adding context labels to provide additional meaning to IP-XACT components. They use this to assess the correctness of interconnections in the system. Strik et al. [6] studied aspects regarding IP (re)configurability to reuse these after a partial change of some parameters. They underline that IP-XACT is missing expression evaluation fields to support flagging illegal (sub)system composition. El Mrabti et al. [7] highlight that in IP-XACT the information used for both Transaction Level Modelling (TLM) and RTL simulations are mingled in the same schema. According to them, this is problematic when Model Driven Engineering (MDE) techniques, e.g. transforming the IP-XACT model to a SystemC simulation, are used. Therefore, they propose an intermediary schema, in which only one type of information is described. This is then further specialized depending on its purpose, e.g. HALports as specialized ports. However, all proposed extensions discussed so far in this section can be seen as level 0 schema descriptions, that consider only the
HW IP block itself. As mentioned in Section I, for systems involving both HW and SW, one also needs to describe the HdS belonging to the HW IP to enable automated integration and generation of the final design. Therefore, we propose level 1 schema descriptions as an IP-XACT extension.

To automatically integrate heterogeneous predefined IP blocks into complex MPSoC architectures, HdS solutions have been proposed. In this context, the software stack comprises the application’s tasks augmented with its associated HdS. Recent research investigated the generation of these stacks automatically to reduce design time. In [14][15][16][17] frameworks are proposed that generate the HdS automatically. However, these approaches target mainly the automated generation of the HdS for predefined IP blocks. This means that the process is restricted only to a set of known IP blocks. This implies that this HdS generation is also predefined, and is equivalent to selecting and configuring the proper software from an available database. This, in turn, restricts the generality of the approach. Instead our solution does not employ a database, since HdS is generated specifically for each custom IP block. This is especially needed in RC systems consisting of both hardware and software parts. Here, the hardware, usually generated with a HILL-to-HDL compiler, e.g. DWARV [9], and the HdS, are custom.

IV. INTEGRATING ORTHOGONAL COMPUTATION MODELS

To investigate the IP-XACT capabilities to model HW/SW co-design tool chains supporting HdS generation and tool interoperability, we used the development of an H.264 decoder application implemented on an Field Programmable Gate Array (FPGA) as a case study. The goal is to integrate different tools and models such that we can automatically generate application specific MPSoC platform implementations of sequentially specified applications. To realize this, we use the Daedalus [10] system level synthesis tool set to implement an MPSoC from sequential C code. In particular, from the Daedalus tool set we use the PN compiler [23] to partition a sequential application and ESPAM [17] to map the partitioned application onto an FPGA. Outside the Daedalus tool-set, we use DWARV [9] to automatically generate hardware IP blocks for performance critical parts of the sequential C code. We first describe the problems observed when integrating the two tools in Section IV-A. Subsequently, we present our extended framework solution based on HdS and IP-XACT meta-data descriptions in Section IV-B.

A. Problem Statement

In [18] the ESPAM-PICO tool was presented, which incorporated the PICO compiler from Synfora Inc. [20] in the Daedalus tool-flow. This approach was used to achieve higher performance for Polyhedral Process Network (PPN)[19] implementations by replacing computation intensive nodes with functionally equivalent hardware IP cores generated by PICO from the available C code. The replacement was done smoothly, as both tools were operating under the same memory model, i.e., a distributed memory model. However, several restrictions were imposed on the C code which can be processed by the PICO compiler. For instance, each loop body could contain only one other loop. Therefore, using PICO as the hardware compiler was not feasible for the H.264 application (Fig. 2) where multiple nested loops are found. DWARV is a compiler which has less restrictions than PICO, making it suitable for generating hardware blocks for our case study.

However, integration of a Custom Compute Unit (CCU), generated by DWARV, in a PPN created by Daedalus is not straightforward. The CCU has an interface suitable for interacting with Molen [2][3] based platforms, which employ a shared memory model. The PPN node, on the other hand, into which the CCU has to be integrated, has multiple input and output FIFO channels typical for the stream-based distributed memory model. Fig. 3 shows the differences in the expected and provided interfaces. The CCU entity expects a common DATA_ADDR bus to access shared memory (Fig. 3 right box), whereas the PPN node has 3 input ports and 2 output ports from/to the distributed memory implementing the channels of the network (Fig. 3 left box). The specific challenge therefore is to find a way to specify the requirements for Daedalus such that DWARV can automatically generate the correct interface.

The following list summarises all challenges encountered in this work:

- **Item 1:** Communicate information between Daedalus and DWARV.
- **Item 2:** Integrate different architectural templates.
- **Item 3:** What tool should generate the HdS.
- **Item 4:** Information that must be exchanged to correctly generate the HdS.

![Fig. 2: H.264 Decoder Application.](image1)

![Fig. 3: Interface Mismatch Problem.](image2)
• **Item 5**: IP-XACT extensions needed to facilitate automated integration.

The IP-XACT standard capabilities for modelling the required tool-flow information exchange, i.e. items 4 and 5 above, are analysed in Section V. Items 1 to 3 are described below, assuming all information to be communicated between tools can be modelled in the standard.

### B. Framework Solution

As shown in the top of Fig. 5, we use the PN compiler [23] of the Daedalus environment to create a PPN from the sequential C code of the H.264 top level function. In Fig. 4, we show a PPN consisting of three processes and three FIFO channels. Each process executes a program that is structured as follows. First, data is read from the incoming FIFO channels. Then, the functional part of the process is executed, which performs a computation on the data read. Finally, data computed by the function is written to outgoing channels. When data is not available in the input channels, the read operation will block while allowing pending computations to finish. When an output channel is full during a write operation, the write operation will block the entire process.

As shown in the left part of Fig. 5, we use ESPAM to subsequently implement the H.264 PPN as a system of point-to-point connected MicroBlaze processors on an FPGA. This means the functional part of each process is implemented as a software program running on a MicroBlaze. Based on profile information given in TABLE I, and DWARV C code compliance rules (i.e., no file system operations), we have decided to accelerate the Inverse Discrete Cosine Transform (IDCT) process using a specialized hardware component. We have used the DWARV C-to-VHDL compiler to generate the CCU from the C code of the IDCT function, which requires the C function to be communicated from ESPAM to DWARV. The HdS layer 1, detailed later in text, is generated by ESPAM and also transmitted to DWARV to provide the HW primitives with which the platform can be accessed. The output of the DWARV tool, which includes both RTL and HdS implementations, is communicated back to ESPAM. In both cases, we use IP-XACT metadata descriptions to facilitate meta-data exchange between tools (item 1).

To solve the interface mismatch problem between DWARV-generated CCUs and Daedalus’ PPNs, DWARV generates a wrapper for the CCU. This wrapper provides memory to the CCU which stores the input/output channel data before/after the CCU is started/stopped (item 2).

The HdS itself is structured into three different layers, that each are concerned only with a part of the functionality needed in the complete software driver. These are platform primitives (HdS layer 1), IP- and OS-specific driver code (HdS layer 2) and an application layer (HdS layer 3). That is, HdS layer 1 provides low-level primitives that are used by HdS layer 2. The primitives in layer 1 strongly depend on the processor the HdS is running on, and the way the CCU is connected to the processor running the HdS. This information is known only by ESPAM. HdS layer 2 provides functions that control the CCU by sending and receiving commands and data to and from the CCU using the primitives provided by layer 1. The separation of HdS layers 1 and 2 makes the HdS layer 2 code independent of the actual platform. HdS layer 3 provides user level functions, which are invoked by a user application to perform the task for which the CCU was designed. The functions in layer 3 solely use functions provided by HdS layer 2. The HdS layer 3 function(s) provide a transparent interface to the CCU, essentially making the CCU available.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Exec. time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>parser</td>
<td>7.17</td>
</tr>
<tr>
<td>cavlc</td>
<td>36.23</td>
</tr>
<tr>
<td>IDCT</td>
<td>29.82</td>
</tr>
<tr>
<td>intra_pred</td>
<td>16.42</td>
</tr>
<tr>
<td>deblock</td>
<td>9.41</td>
</tr>
<tr>
<td>main</td>
<td>0.95</td>
</tr>
</tbody>
</table>

Table I: Profile Information for H.264.
as a regular software function.

The integration of the whole system is depicted in Fig. 5. The tool flow is initiated by the PN compiler, which generates a PPN. Then, an IP-XACT package is created containing the C code that is to be synthesized into hardware and the layer 1 HdS generated by the ESPAM tool. The DWARV tool uses the C function to generate the CCU and the layer 1 HdS to generate the remaining layer 2 and 3 HdS (item 3). Subsequently, these are described in IP-XACT and sent back to the ESPAM tool which extracts the required meta-information and compiles all HdS into an executable. Also, ESPAM integrates the CCU RTL into the system.

V. IP-XACT EXTENSIONS

In this section, we elaborate on the expressiveness of the current IP-XACT standard to describe the scenario presented in the previous section. Based on this analysis, we describe three possible extensions, namely hardware compiler input, HdS and tool-flow integration related extensions. We illustrate the proposed extensions using the vendorExtensions extension construct that is already part of IP-XACT. This allows vendor specific information to be added to IP-XACT descriptions.

A. Hardware Compiler Input

As mentioned in the previous section, to accelerate a process of the PPN we use the DWARV compiler. DWARV accepts a C function as input and generates a VHDL component that implements the original C function. In our case study, we send the C function together with an IP-XACT metadata description to DWARV. DWARV assumes that a function argument can be both input and output at the same time, if the argument is a pointer into shared memory. However, arguments to functions inside a PPN process are always unidirectional, that is, either input or output. For each function argument, unidirectional First In, First Out (FIFO) channels are created according to the process network topology. Therefore, we need to inform DWARV about the direction of each function argument, such that the appropriate FIFO input or output connection can be generated. We therefore add a field to the IP-XACT description that is passed along with the C file defining the function implementation. In Fig. 6, we show how this direction field can be added using vendorExtensions. In our scenario, DWARV supports three different directions, namely in, out and inout. The latter will not be used by the Daedalus environment, but might be used by other tools interfacing to DWARV.

B. Hardware Dependent Software

Using HdS in heterogeneous MPSoC systems abstracts hardware and OS details away from the user or application level. In our case study, we have partitioned the HdS into three different layers, as described in Section IV-B. HdS layer 1 is generated by the Daedalus environment which has full knowledge about the hardware platform, the mapping of processes onto processor components and the interconnections between processor components. Therefore, Daedalus ensures that HdS layer 1 uses the correct hardware platform primitives to interface the CCU. For instance, for one processor these primitives use memory-mapped I/O, whereas for another processor dedicated instructions are available. The generated HdS layer 1 is described in IP-XACT which is passed to the DWARV environment. This enables DWARV to generate HdS layers 2 and 3 that make use of the primitives provided by HdS layer 1.

The right side of Fig. 7 shows the HdS hierarchy along with their corresponding generated functions. To create a semantic link between two different HdS layers, we need to specify the purpose of the functions found in HdS1. For HdS layer 1, we classify a function as read, write or command. An example of such a description in IP-XACT is shown in Fig. 8. The read identifier classifies the function as one that reads data from the CCU-Processor Interface (CPI), which has been implemented using two small FIFO buffers. The write identifier classifies the function as one that writes application data to the CPI and the command identifier classifies a function as one that
writes control data to the CPI. Because hardware primitives are typically limited in number, we define a new IP-XACT type, i.e. Hds type, so that a tool can automatically decide which function should be used when Hds of a higher layer is generated. With only the above three types defined, we create a semantic link between layers 1 and 2.

Similarly, we can create a link between layers 2 and 3. However, layer 2 is concerned with abstracting OS specific implementation details for the custom IP block, and since there is no OS present in our case study, we leave the definition of this type as future work. Nevertheless, we can imagine that this type could include identifiers for the POSIX standard [22] such as opening and closing file handles.

C. Tool Chains

To fully automate the tool flow shown in Fig. 5, IP-XACT provides means to model generator chains. For example, the current IP-XACT standard provides a generatorExe field which contains the executable to be invoked for a generator. However, we observe that IP-XACT currently lacks a way to describe the tool-specific configuration files. For example, Dwarv uses an external Floating Point (FP) library description file listing the available FP cores such as FP addition, so that floating point arithmetic in the C code can be implemented using available FP cores. To allow seamless cooperation of different tools from different vendors, we observe the need to include tool-specific descriptions and files in IP-XACT generatorChain schema. Fig. 9 shows an example generator specification of Dwarv in IP-XACT. Whenever the fileset is present, the parameter list when invoking Dwarv will include the filesets specified in the vendorExtensions field, e.g., FPlib_v1.2.fpl.

VI. IMPLEMENTATION

In this section we provide implementation details required for tool interoperability for the case study presented in Section IV. Based on the implementation steps followed, we explain how to bridge the interface mismatch problem and show that without tool automation these issues limit design productivity. We start with the extensions required in the ESPAM tool and subsequently discuss the extensions required in the Dwarv compiler.

A. ESPAM

The ESPAM tool is split into two distinct phases. Namely phase 1, in which a package is generated for each PPN node that is mapped onto a CCU. As shown in Fig 5, this package contains the C code that has to be synthesized using Dwarv, a meta-data description of the C code (cf. Section V) and a meta-data description of the Hds 1 primitives. In phase 2, the conventional FPGA backend of ESPAM is run such that an XPS project is obtained. Then, this XPS project is modified according to the IP-XACT generated by Dwarv. This means HDL and Hds files generated by Dwarv are put in the appropriate locations and the CCU is connected in the hardware specification (.msh) file.

B. Dwarv

The Dwarv compiler employs the CoSy Compiler Development Platform [21]. The compiler is composed of engines which work on the Intermediate Representation (IR) of the input program.

On the left side of Fig. 10 we show the main building blocks of a hardware compiler. To generate RTL from a HLL one will need to create a Control Data Flow Graph (CDFG) of the input C code (CDFG Building), schedule it (CDFG Scheduling), and emit the VHDL (VHDL Generation). Some of the most important engines that compose these basic blocks are shown on the right side of the picture. The engines in bold represent custom written engines, specific for hardware generation, whereas the other ones are standard CoSy engines. This last category implement common transformations and optimizations of the IR, such as conversion into Static Single Assignment (SSA) and Common Subexpression Elimination (CSE).

For this work we have implemented two more engines. The first one parses the IP-XACT received from ESPAM, extracts all the information needed and stores it in the IR. The second engine queries the IR for specific meta-data such as the corresponding function parameters and, based on application specific requirements, generates the Hds accordingly. Our case study also required a hardware wrapper to emulate the shared memory model, which is also generated. Fig. 11 shows the
TABLE II: Experimental Results.

<table>
<thead>
<tr>
<th>Design</th>
<th>Slice Usage</th>
<th>Exec. time (cycles)</th>
<th>Speedup (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>All-MicroBlaze</td>
<td>2727</td>
<td>273806856</td>
<td>0</td>
</tr>
<tr>
<td>DWARV-IDCT</td>
<td>7473</td>
<td>209958437</td>
<td>30.4</td>
</tr>
</tbody>
</table>

original interface of the CCU, whereas in Fig. 12 the updated interface available to the outside system after the tool has been modified is shown.

VII. EXPERIMENTAL RESULTS

In this section, we report on two kinds of results. First, we show the applicability and usefulness in a real-world application and second, we report the overall productivity gain. We base this on our experience with the H.264 case study, for which the first implementation was done manually.

A. Validation of Approach

In our experiments, we target a Xilinx Virtex-5 XC5VLX110T-2 FPGA and use Xilinx EDK 9.2 for low level synthesis. The video resolution we use in the experiments is QCIF (176x144 pixels). To validate the approach, we implement the H.264 decoder application twice. The first time we map all processes of the PPN onto MicroBlaze processors, which means all PPN processes are implemented as software. This serves as our reference implementation. The second time we replace the software IDCT node with a hardware version obtained using the methodology described in the previous sections. We obtain a speed-up of approximately 30.4%. The results are summarized in TABLE II.

Consider Amdahl’s law:

\[
\frac{1}{(1 - P) + \frac{P}{S_P}} = S_{overall}
\]

where \( P = 0.2982 \) represents the IDCT percentage of the whole H.264 decoder execution time and \( S_{overall} \) the overall application speed-up. Substituting 1.304 for \( S_{overall} \) and solving the equation for \( S_P \) gives the individual speed-up of the IDCT function of 458%. Given the maximum achievable application speed-up when optimizing only the IDCT process, i.e., 42.5%, we can deduce that we are at 71.5% application-wise acceleration of the theoretical maximum. This shows that good results can be obtained by integrating tools targeting different architectural templates.

B. Productivity Gain

Besides proving the usefulness of the approach to obtain a faster implementation of a PPN, we discuss the productivity gain observed when adopting an automated IP-XACT based approach. If the automated support was not available, manually patching the tools would have been time consuming and error-prone. Depending on the application knowledge of the system designer and application complexity, activities like writing the HdS or the CCU wrapper can take from a few hours up to even weeks. Moreover, validation may take a similar amount of time. For example, a memory map has to be specified as C code in the HdS and as VHDL in the RTL. For correct operation of the system, these two representations need to be fully consistent, which may be an important source of errors when manual action is involved. We eliminate such errors by taking the information from a central source (e.g., an IP-XACT description) and then automatically generate the different representations. This substantially reduces the time needed for validation. To fully understand the specific challenges and properly design the modifications required by the tools to enable automated integration, our first implementation of the system was manual. Based on this experience, we estimate the total time required to build a fully working system for the H.264 decoder application. The estimated times are summarized in TABLE III for the main activities we performed. This shows for example that the most demanding task, i.e., writing the CCU wrapper, has been reduced from 3 days to approximate 4 seconds. Summing up all these individual efforts, and comparing against the time needed to automatically generate the whole design, we conclude that the productivity gain for this case study is about one person-week.

VIII. CONCLUSION

In this paper, we have presented a new approach for automated generation of RTL implementations from sequential programs written in the C language. This is achieved by combining the Daedalus framework with the DWARV C-to-VHDL compiler with the aid of the IP-XACT standard and HdS solutions. The integration experiment showed that with these concepts, even different architectural templates can be reconciled. Furthermore, we investigated the capabilities of the IP-XACT standard to model automated integration of MPSoCs consisting of both hardware and software components. The components of the system originated from different sources, e.g., libraries and tools, which makes automated integration a non-straightforward task. Our work was inspired by a case study of a H.264 decoder. In particular, we focused on the scenario where a hardware component needs to be controlled by a software component. We found that the Hardware Dependent Software needed to control the hardware component cannot be described in the current IP-XACT standard. We identified three possible concepts that could be added as extensions to the IP-XACT standard to realize automated integration of HW/SW systems. Using these, we were able to compose, integrate and generate the complete system described in Section IV in a Design Environment. The speed-up obtained was 30.4% application-wise.
entity CCU_Idct is
port (...
end CCU_Idct;

entity CCUwrapper_Idct is
port (...
end CCUwrapper_Idct;

Fig. 11: Original CCU.

Fig. 12: Generated CCU Wrapper.

REFERENCES


