Abstract—IP-XACT, the recent IEEE1685 standard, defines metadata format for IP packing and integration in System-on-Chip designs. It was originally proposed for hardware descriptions, but we have extended it for software, HW/SW mappings and application communication abstraction. The latter is realized with Multicore Association MCAPI that is a lightweight message passing interface. In this paper we present as a work-in-progress how we utilize all these to deploy and move application tasks between different platforms for FPGA prototyping, execution acceleration or verification. The focus is on the metadata format since it is a foundation for automation and tool development. The design flow is illustrated with two case studies: A motion JPEG encoder and a 12-node workload model of video object plane decoder (VOPD). These are deployed to PC and Altera and Xilinx FPGA boards in five variations. The results are reported as the deployment time for both non-recurring and deployment specific tasks. Setting up a new deployment is a matter of hours when there is an IP-XACT library of HW and SW components.

Index Terms—electronic design automation, multiprocessor system-on-chip, meta-data, IP-XACT, IEEE1685, MCAPI, Multicore Association, Kactus2, deployment

I. INTRODUCTION

Current System-on-Chip designs include many Intellectual Property (IP) blocks, currently about 50 IPs in total and ten processor cores. The amount of IPs increases steadily about 20% per year. At the same time, the amount of embedded software increases even more rapidly. Despite the SoC design itself, the challenge is also the overall product design especially for industrial embedded systems. Today 60% of the SoC design should be reused and productivity increased 10x by year 2022 [1].

One of the most acute problems is how to quickly deploy a product to a new platform either because it offers better performance, cost reduction or because the old one is discontinued. The same challenge is inherently present in design verification, e.g. in FPGA prototyping and emulation. In addition, reuse often proves complicated in practice although all designers recognize its importance and benefits.

SoC design tools are principally categorized as creation and integration environments [2]. The former ones start from high abstraction level models, and transform the models to executable codes for supported platforms. In this paper we focus on the latter approach, which includes modularizing, reusing and putting together components from libraries.

Specifically, we focus on abstracting and describing the components, designs and design configurations in XML metadata. The practical goal of the metadata is to ease compatibility of design automation tools. However, in this paper we exclude the automation and show how the metadata is used in each design step. Two case studies demonstrate the flow and metadata usage for five different deployments on FPGAs. The performance of the methodology is measured in terms of design effort, time, and system performance.

Our approach is based on IEEE1685/IP-XACT metadata standard [3], which is originally purposed for hardware IP-block integration. We extend IP-XACT to SW components and HW/SW mappings, as well as application communication abstraction using Multicore Association Communications API (MCAPI) [4]. The benefit is the ability to describe all SoC components in a uniform, standard way, and include also the design process and tools related information. Throughout the deployment flow, we use freely available Kactus2 IP-XACT tool [5] [6] and IP cores [7].

The key contributions in this paper are:

- New extensions to IP-XACT 1.5 for MCAPI, SW and HW/SW mappings
- Two design examples using the extensions
- New MCAPI platform-dependent transport layer implementations for shared memory and message passing architectures
- Design effort measures for changing SoC deployment between different platforms

The paper is organized as follows: Section II discusses related works and Sections III and IV introduce the design flow and the case studies. Section V briefly explains our IP-XACT extensions. Section VI gives details how metadata is used in the flow. Sections VII and VIII present the results and conclusions.

II. RELATED WORK

Any system design includes lots of source files (VHDL, C/C++), HW and SW libraries, tool specific project files and target specific executable files. A project can consist of 10k files on hundreds of folders. Traditionally the dependencies
between parts are in the form of file references and path definitions. Care must be taken that the paths remain valid. One major problem is how to quickly apply changes in platforms, applications or both.

Version management alone does not solve the whole problem, since it is immune to the meaning of the files. Improvements include database-based system descriptions and use of metadata, most often in XML or script languages like TCL. Several SoC design frameworks include domain-specific and custom metadata formats in companies and research frameworks like Koski [8] and Daedalus [9]. Recently, IP-XACT extensions are proposed to exchange application description from process networks to a reconfigurable computing HW generation in DWARV (Delft Workbench Automated Reconfigurable VHDL) [10]. However, their approach is to fit IP-XACT to their domain, which reduces general applicability of IP-XACT.

Xilinx has presented a big set of vendor extensions, in total of 119 different XML elements. Extensions are for their own tool support and new attributes and dependencies e.g. for supported devices, IP taxonomy and port endianness. Xilinx ISE Design Suite 13 includes 50 Xilinx IP cores and by end of 2012, all Xilinx IP cores will support IP-XACT [17].

Although vendor extensions are handy when working with one tool only, they create vendor lock-in. Consequently, design data is harder to port from tool to another, and requires transformation tools.

As far as we know, we present for the first time IP-XACT extensions for explicit SW components, MCAPI nodes, endpoints, channels and HW/SW mappings. It should be noted that we do not present a principal new system model but a metadata format that reuses IP-XACT 1.5 constructs.

MCAPI was chosen since it aims at embedded and many-core systems with lightweight and efficient implementations. One principal requirement was support for platforms with and without an operating system, as well as abstracting HW blocks with the same abstraction level. MCAPI was found as a suitable standard. Since MCAPI is quite a new standard there are not many publications comparing it to other parallel APIs.

III. DESIGN FLOW OVERVIEW

Figure 1 depicts an overview of the proposed deployment flow. The left side (A) shows the phases with an example design and the right side (B) the IP-XACT metadata objects used during the flow. Our starting point is executable application description, and now we do not consider how this is obtained. An example is a set of C source codes that should be deployed to a new platform and possibly some of them to HW acceleration. The application can be complete or a model representing external behavior.

The next step is grouping of the application modules according to mutual communication by performing static analysis or with simulation traces. For example, highly data dependent modules are put together. For such modules the next step is abstracting the communication using MCAPI nodes and endpoints. MCAPI channels connect the endpoints and are illustrated as dotted lines in Figure 1A. This step makes the application modules independently portable. Abstract communication incurs some overhead and therefore it is not beneficial to convert all function calls into MCAPI endpoints. Appropriate granularity depends on both the application and the underlying platform.

The third step includes decision what is executed on HW and what on SW by means of automated or manual architecture exploration. Accelerator modules for HW execution have the same MCAPI endpoints as their SW counterparts, so it does not matter which implementation is selected. The HW implementation e.g. in VHDL is reused either from a library or a newly created at this step.

In the last step, the application modules are deployed to the desired platforms, including e.g. several different FPGAs, boards and computers. The deployment can be the final product implementation or prototyping and verification. Independent of technologies, the platform must include MCAPI protocol layers for processors. For accelerator HW IP blocks the MCAPI endpoints can be virtual (access included in processors MCAPI implementation) or directly implemented in HDL at the IP-block. The final outcome is quick change of the deployment between different platforms.

In the proposed design flow, Kactus2 is used in all steps that need metadata capture. The main principle is to use the standard IP-XACT elements like components, objects and bus interfaces as far as possible to help tool development and compatibility. The right-hand side of Figure 1 lists the
deployment flow related objects (IP-XACT component, Design) and the extensions defined as explicit relations (VLNV reference).

IV. HANDLING SW AND COMMUNICATION

In IP-XACT, component is the main object type and it describes one reusable IP, for example CPU, memory or accelerator. Its internal structure can be captured with design that contains instances of other components and their connections. In a sense, the component corresponds to a VHDL entity and the design to architecture. Bus definitions groups several I/O pins together and simplifies making connections between components. All IP-XACT objects are referenced using a unique VLNV (Vendor, Library, name, Version) to build up systems, which means that the file names or locations on disk do not matter.

Increasing fraction of SoC designs regard SW, but currently IP-XACT considers mostly HW. However, many aspects are very similar in SW as well and hence IP-XACT is directly applicable, for example SW modules correspond to HW components. Moreover, we propose the following extensions to describe, combine and configure SW building blocks, as well as bind them to HW resources.

A. Referring software from IP-XACT objects

IP-XACT components have one or multiple views, for example a hierarchical design with references to other component VLNVs or a flat one with references to files as shown in Figure 2. IP-XACT design is a separate object and it is referred to in component’s view.

In addition to referring to a SW file directly from a HW component, we extend IP-XACT to refer to a new SW design. Unfortunately, that is by default component-specific and not instance-specific, just like the file sets and the object reference to a HW design. Therefore, having multiple identical CPUs (HW instances) executing different SW is cumbersome. Either one needs to add a new view for each SW for the HW component, or make a new HW component version for each SW. The former blows up the number of views in the HW component, and the latter the number of HW components in a library. Thus, SW for CPU should also be stored to the HW component instances.

Alltogether we have three choices for referring to SW from a HW component: i) the standard way: fileset references per CPU for SW, ii) we add a custom parameter for each CPU to store the VLNV reference to a SW design, or iii) we store the mapping information into a separate, new design type. We call the latter a system design that we explain later.

B. Capturing SW stack structure

We follow the general model of a SW stack, which includes a platform independent application layer on top and a SW platform below it having a hardware dependent layer at the bottom. When designed properly, changing one layer does not require changes on other layers. SW platform as whole should be application (case-specific) independent.

In our SW design, we can describe the structure of SW as a SW stack (with the layers), or without it just to show how SW components are connected together. Figure 3 shows the simplified model of the SW design, in which instances are denoted with i0, i1…
only references to the needed files, and a general interface that
does not specify any API.

The benefits of SW components and designs are formalized
documentation, better library management and potential to
add automatic checking and compilation.

C. Communication abstraction with MCAPI

Structural packetizing of the software helps in the design
management but does not reveal the inter processor
communication patterns in the system. Structural description
tells what parts are connected but not how they communicate
as the exact details are “hidden” in the source codes.

For example MCAPI is very suitable for alleviating this. It
abstracts the system as nodes (e.g. thread) that have multiple
endpoints. Consequently, the top-level view of the system is a
set of interconnected nodes, regardless of the HW hierarchy
level they reside. These nodes are usually processors, e.g.
inside a workstation, on a PCB board, or on an FPGA, and all
are shown together. Moreover, in our work HW accelerators
can be accessed using the MCAPI endpoint abstraction instead
of e.g. register addresses. The MCAPI channels between
nodes’ endpoints define which nodes communicate with each
other.

There are three types of unidirectional endpoints: scalar,
message, and packet. For example, a DCT function may have
two inputs (pixel data and quantization parameter) and one
output (frequency coefficients). Many functions can be easily
chained. The same DCT can be reused in many systems by
changing the connections between the endpoints, e.g. are the
pixels coming from a camera or from a processor.

Obviously, MCAPI function calls must be translated to the
underlying platform, for example as calls to Ethernet driver on
a FPGA [14] and POSIX calls on Linux PC. Implementation
of this MCAPI transport layer is part of the SW platform
stack, as depicted in Figure 3. Such abstraction simplifies
making different deployments with modest effort.

Explicit description of MCAPI nodes is implemented as IP-
XACT bus interfaces for SW components. However, we can
also define virtual endpoints for HW components by
introducing special communication interfaces for them. MCAPI
channels are implemented as IP-XACT interconnections between
bus interfaces, and defined in our system design.

To help importing code, Kactus2 tool can analyze the
legacy source codes looking for endpoints and then add them
to the IP-XACT SW component automatically.

D. Separate system design phase for mapping

In the simplest cases component-specific SW has been
associated to a HW component already during the IP
packetization, for example in case of a fixed application-
specific processor that always computes the same task in all of
its instances. However, in general, we must have a way to
specify instance-specific SW for each CPU. Therefore, we
create a separate system design which tells the mapping
between SW and HW instances as depicted in Figure 4.

The upper level component, e.g. “FPGA-based product”,
refers to the HW and system designs. This way we can
introduce several SW mappings to the same HW or the same
SW for several HW platforms.

Figure 4 System design describes instance-specific mapping of SW to HW in
IP-XACT.

Our system design shows graphically a flattened HW
design that tells what SW instances are mapped to which HW
instances. Each CPU instance (a programmable component),
and HW instance with virtual MCAPI endpoints, is
automatically imported to the system design by Kactus2. Then
the user drag-and-drops SW components from the library onto
the HW placeholders, or creates new SW components from
scratch for the HW instance.

Unfortunately, there is no native way in IP-XACT to store
references to instances, but only to VLNV which always
points to a library object. Hence, we add vendor extensions.
System design is now linked to a HW design and each SW
component instance is linked to its host HW component
instance, as shown in Figure 4. It should be noted that only
HW components with CPUs or virtual MCAPI endpoints are
available for the SW mapping.

Each CPU placeholder contains all application and SW
platform components executed by that CPU. If IPC ports, such
as MCAPI endpoints, have been defined, the designer
connects them as well to visualize which application parts
communicate. These system-wide MCAPI channels are
defined as IP-XACT interconnections and bus interfaces
between SW component instances as depicted in Figure 5.
HW instance i0 is of type CPU and includes SW components
with MCAPI endpoints. i1 is a fixed HW components with
virtual MCAPI endpoints.

Creating MCAPI channels is not only for documentation,
but it is also used in writing the application code in Kactus2.

The code editor suggests auto-completion of valid system
wide endpoint identifiers and other MCAPI related tasks.

During the SW generation the system-wide unique endpoint
IDs are automatically generated by Kactus2. It also checks
automatically the endpoint types and that the communicating
parties have also a physical connection.
system design. The main modules are a design manager and a library handler that are also visible in GUI. The former takes care of the main tasks, including design capture and invoking generators. The library functions parse XML metadata into C++ classes for the manager. They also provide editors for adding and modifying IP-XACT objects.

Kactus2 includes moderate design automation by itself, e.g. object validation, tracking of changes and built-in generators for VHDL, Quartus, Modelsim simulation, MCAPI code template and header file generation. For example, VHDL generator takes hierarchical IP-XACT component as input and creates a structural top-level VHDL description from it. Architecture exploration, automated HW/SW mapping and other system level automation can be implemented as standard IP-XACT generators.

B. Kactus usage

In this paper, Kactus2 has three main roles. First, it is used to packetize HW and SW platform and application components, including MCAPI endpoints. The objects include generic parameters, associated files and information about the tools used and their languages and options.

The second use of Kactus2 is to create the designs: hierarchical deployment-specific HW designs, application independent SW stacks, SW designs to be associated with HW components, and system designs that collect all parts together. All of these are designs carried out manually in this paper. However, generators are used to create implementations on FPGAs.

Third, Kactus2 also helps to manage the library and show dependencies between the objects, both for user and generator created ones. This is the more important the more deployment experiments and versions exist [16].

VI. CASE STUDIES

The proposal is demonstrated with two case study applications and three platforms. Case studies consider how the application modules are described and moved from PC to different FPGA platforms. Given the limited space and the complexity of the topic, simple toy applications are used here for demonstration.

A. Applications

The first case is an intra-frame video encoder (motion-JPEG) given in ANSI C-code. The original encoder uses shared memory architecture and runs on a single thread. We modify the code so that the DCT transform could be more easily executed on another processor or a specialized HW accelerator.

The second case is a workload model for a Video Object Plane Decoder [17]. It is a process network including 12 nodes, 14 arcs and a timer, as shown in Figure 6. Each graph’s node is implemented as a MCAPI node and arcs as MCAPI channels. Since this is a coarse model, it shows only communication performance. However, we can easily modify the ratio between computational load and the amount communication. The tasks wait for input tokens (certain number of bytes) before they are triggered for execution.

Here, “execution” means just waiting since this is a...
workload but not a functional model. After execution, a task sends one or more tokens to other tasks.

Figure 6 The task graph of VOPD.

B. HW platforms

Our physical hardware platform consists of a PC, Altera DE2 and Xilinx Spartan-3E FPGA boards. FPGAs accommodate soft NIOS and uBlaze processors, DCT as an exemplary HW IP-block and Avalon, HIBI and Xilinx PLB on-chip buses as well as Ethernet interfaces. NIOS processors also included HIBI DMA connected to Avalon and dual-port memory as in [14]. Simplified structure of the HW platform is shown in Figure 7. In the experiments, the FPGAs were run at 50 MHz. The logic utilization was few thousand LUTs.

Figure 7 HW platform with three different types of CPUs.

C. SW platform and MCAPI implementations

The software platform includes MCAPI transport layers for Linux (PC), uC-Linux (NIOS) and a non-OS version (NIOS, uBlaze), Ethernet stacks and HIBI bus macros and drivers. PC uses Debian GNU/Linux 6.0 and NIOS uC-Linux ver. 2.6.

OpenMCAPI is used on PC and on Nios that runs uC-Linux. For Nios and uBlaze without any OS, MCAPI reference implementation from Multicore Association (MCA) was modified to use only message passing. It allows only one MCAPI node per processor since no scheduler is available. Furthermore, the channels are created statically. The implementation is very concise, just less than 5k lines of C. It took few weeks to implement. It should be noted that a MCAPI port is a non-recurring task.

A special feature is that HW accelerators are seen as virtual endpoints [14]. Other nodes can use them similarly regardless of whether they are executed on SW or HW. For all virtual endpoints, we need and adapter that transforms MCAPI transactions to native device operations. In this case this was done for a DCT IP-block device driver.

Figure 8 depicts the protocol stack used in PC, NIOS and uBlaze. OpenMCAPI uses POSIX and shared memory model for MCAPI. In heterogeneous platforms we may need also message passing, so we implemented a topology mapping layer into MCAPI transport. In practice, we have a header file describing where MCAPI nodes are mapped. The header is generated by Kactus2 from the system design. Based on the node ID, the mapper selects whether to use shared memory or message passing.

VII. RESULTS

Table 2 summarizes the overall complexity in terms of files and lines of code for the case studies. We notice that MCAPI is the largest software module in terms on lines. Realistic applications are easily an order of magnitude larger and hence the overhead from MCAPI is not as drastic as here.

Table 2 Case study components without metadata files.

<table>
<thead>
<tr>
<th>Item</th>
<th># of Files</th>
<th>Lines of code (language)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video enc. (orig, Socket, MCAPI)</td>
<td>8</td>
<td>C:&lt;1k</td>
</tr>
<tr>
<td>VOPD (MCAPI)</td>
<td>12</td>
<td>C:&lt;10k</td>
</tr>
<tr>
<td>OpenMCAPI transport, Linux</td>
<td>77</td>
<td>C:4.5k</td>
</tr>
<tr>
<td>OpenMCAPI transport, uC-Linux</td>
<td>75</td>
<td>C:4.5k</td>
</tr>
<tr>
<td>MCAPI transport, no OS</td>
<td>5</td>
<td>C:1.5k</td>
</tr>
<tr>
<td>HIBI driver NIOS</td>
<td>3</td>
<td>C:2k</td>
</tr>
<tr>
<td>Xilinx lwIP</td>
<td>40</td>
<td>C:2.5k</td>
</tr>
<tr>
<td>uC-Linux</td>
<td>55k</td>
<td>C:18M</td>
</tr>
<tr>
<td>DCT accelerator</td>
<td>7</td>
<td>VHDL:3k</td>
</tr>
<tr>
<td>HIBI PE DMA controller</td>
<td>5</td>
<td>VHDL:2k</td>
</tr>
<tr>
<td>HIBI segment</td>
<td>15</td>
<td>VHDL:7k</td>
</tr>
</tbody>
</table>

It is not easy to measure the benefit of reuse and abstraction, or design methodology in general. The primary metrics here is the required user effort in terms of wall-clock time as well as number of files and code lines. The former varies greatly between individuals whereas the latter ones provide a first-order approximate for comprehension complexity.

A. Performance results

It should be noted that performance is not the primary objective in this paper. However, the MCAPI cost was measured from the video encoder running on PC. The calls to the most time consuming function, DCT, were modified in four ways.
First, the pass-by-reference was converted to pass-by-value, i.e. using memcpy() for the input data and results. This increased the runtime only by 2%. This is the lower bound overhead if DCT would be executed by an accelerator (or another processor) that does not have access to the processor’s data memory.

Then, the DCT and main functions were turned into separate threads and communication is done via shared memory. The runtime increases by 43% since it turns out that system calls take quite a large fraction of time in this case. Note that the main function was just waiting for DCT results instead of a pipelined execution. Last, the communication was done using Sockets and MCAPI calls using OpenMCAPI. Both cases used two threads. The overheads were measured by Intel Vtune. As expected the runtime increased, up to 3x of the original. The overhead from threads is the same as before but there is large amount of bookkeeping associated with the communication abstraction. This was surprising, especially because our earlier experiments on Nios had extremely low overheads using our modified MCAPI implementation from MCA that was just tens of cycles per call [14]. However, OpenMCAPI uses lots of Linux system calls and the overhead was nearly 20 kcycles per transfer.

This result shows that further optimization is required to gain both reusability and performance at the same time for openMCAPI.

B. Deployment effort

The MCAPI communication abstraction and other modifications to the video encoder were very straightforward, just about half a working day for an engineer already familiar with the original version. In other words, converting a program into using MCAPI proved to be efficient. The total number of code lines increased only about 8%. In practice, only two files were modified (main.c and dct.c). Similarly, implementing the workload model of VOPD took only two days for an engineer without any earlier experience from MCAPI.

Another part is the effort needed to create the IP-XACT metadata descriptions, and most importantly designs that reflect the different deployments.

The video encoder experiment has the following use cases:

1) Run everything on PC
2) Run the DCT accelerator on FPGA and the rest on PC
3) Run the DCT on Nios-processor and the rest on PC

We capture all the needed metadata for the case in Kactus2. At least three HW designs are needed: PC+FPGA board, structure of the FPGA board, and internal structure of the FPGA. The needed components include: PC, board, FPGA chip, memories, Ethernet chip, and all board peripherals. All components need also bus interface definitions. Table 3 summarizes the number of files and time associated to Kactus2 usage.

<table>
<thead>
<tr>
<th>Metadata type</th>
<th>MJPEG #Files</th>
<th>VOPD #Files</th>
<th>Time</th>
<th>SW component</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW component</td>
<td>20</td>
<td>20</td>
<td>4</td>
<td>+5</td>
</tr>
<tr>
<td>HW interface</td>
<td>12</td>
<td>12</td>
<td>2</td>
<td>+2</td>
</tr>
<tr>
<td>HW design</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>+2</td>
</tr>
<tr>
<td>SW component</td>
<td>2</td>
<td>2</td>
<td>0.2</td>
<td>+1</td>
</tr>
<tr>
<td>System design</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>41</td>
<td>41</td>
<td>8.2</td>
<td></td>
</tr>
</tbody>
</table>

Table 3 Created metadata files and the time in hours spent on creating them.

Most of the HW-related metadata was already in the library, shown on top of the table. Application and mappings are case-dependent and created from scratch, shown on the bottom of the table. Moreover, we took a ready-made DCT HW IP-block described in VHDL, and created an IP-XACT HW component and associated to it a virtual MCAPI endpoint. Packetizing a working HW component is a straightforward task and takes about 10 minutes. Reusing the DCT IP-block in another SoC is just a drag-and-drop operation, a matter of seconds.

Application is divided in two parts: DCT and the rest as two applications and two endpoints. The deployments are carried out for two platforms according to the three use cases.

Table 4 Video encoder deployment time [hours].

<table>
<thead>
<tr>
<th>MJPEG phase</th>
<th>PC</th>
<th>DCT@HW</th>
<th>DCT@NIOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCAPI comm. abstraction</td>
<td>4</td>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>IP-XACT library setup (HWP/SWP)</td>
<td>1</td>
<td>+6</td>
<td>+1</td>
</tr>
<tr>
<td>IP-XACT HW design</td>
<td>0.2</td>
<td>+1</td>
<td>+0.2</td>
</tr>
<tr>
<td>IP-XACT System design</td>
<td>0.2</td>
<td>+0.2</td>
<td>+0</td>
</tr>
<tr>
<td>Generators (VHDL, C)</td>
<td>-</td>
<td>+0.2</td>
<td>+0</td>
</tr>
<tr>
<td>Synthesis and compilations</td>
<td>0.2</td>
<td>+0.8</td>
<td>+0</td>
</tr>
<tr>
<td>Total</td>
<td>5.6</td>
<td>+8.2</td>
<td>+1.2</td>
</tr>
</tbody>
</table>

Table 4 and Table 5 summarize the deployment effort in hours. For the MJPEG, the first column reports time to PC only deployment. To move DCT to HW on Altera FPGA, most of the extra time is required for the library setup. It included packetizing such HW and SW platform components that are not dependent on any application. To move DCT from HW to NIOS and uCLinux requires only NIOS related HW and SW platform components to be packetized (+1 hour). In summary, The PC deployment took about 6 hours, moving DCT to FPGA about 8, and after that moving the DCT to NIOS 1 hour. If the library were complete in the beginning, all deployment times would have been about 1 hour.

The VOPD case included deployments on PC and a combination of PC, DE2 and Spartan boards. Again we
measure time to set up everything from scratch for PC and report the difference when FPGAs are involved.

Table 5 VOPD deployments time [hours].

<table>
<thead>
<tr>
<th>VOPD phase</th>
<th>PC</th>
<th>PC+ALTERA+XILINX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation, incl. MCAPI abstr.</td>
<td>16</td>
<td>+0</td>
</tr>
<tr>
<td>IP-XACT library setup (HWP/SW platforms)</td>
<td>9</td>
<td>+6</td>
</tr>
<tr>
<td>IP-XACT HW design</td>
<td>2.5</td>
<td>+1</td>
</tr>
<tr>
<td>IP-XACT System design</td>
<td>0.5</td>
<td>+0.2</td>
</tr>
<tr>
<td>Generators (VHDL, C)</td>
<td>-</td>
<td>+0.2</td>
</tr>
<tr>
<td>Synthesis and compilations</td>
<td>0.2</td>
<td>+1.8</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>28.2</strong></td>
<td><strong>+9.2</strong></td>
</tr>
</tbody>
</table>

Figure 9 Kactus2 v1.3 system design of VOPD case study with 4 CPU instances, 12 application tasks and SW platforms.

Figure 9 shows a screenshot of Kactus2 system design of the VOPD case. Independent of the HW hierarchy, the system design flattens all components that can accommodate MCAPI endpoints. Thus, Figure 9 shows a PC, two NIOSes and one uBlaze as hardware components. Each has a SW platform stack, the endpoints and associated application code. MCAPI channels show clearly that PC communicates only with the first Nios, but not with the other or with MicroBlaze.

VIII. CONCLUSIONS

We presented a proposal to deploy applications on MCAPI communication abstraction, and how IP-XACT metadata is extended to handle it. The case studies illustrate the effort when no automation is involved. Packetizing of the HW and SW components, and the HW and system designs with MCAPI definitions are carried out manually. The figures of effort included both non-recurring tasks and deployment specific, recurring effort. Provided a solid IP-XACT library, deployments can be carried out in less than an hour of user time. Our future work includes introduction of non-functional property extensions and IP-XACT generators to help automating the deployments.

**REFERENCES**


