Seamless integration of HW/SW components in a HLS-based SoC design environment

Tiago Rogério Mück and Antônio Augusto Fröhlich
Software/Hardware Integration Lab
Federal University of Santa Catarina
Florianópolis, Brazil
Email: {tiago,guto}@lisha.ufsc.br

***Abstract*—With *system-on-chip* (SoC) designs growing in complexity, system-level approaches that leverage on *high-level synthesis* (HLS) techniques are becoming the workhorse of current SoC design flows. In this scenario, we propose a component communication framework that allows for the seamless integration of hardware and software components in a HLS-capable environment. The proposed infrastructure relies on C++ static metaprogramming techniques to efficiently abstract communication details in high-level C++ implementations of components. We show how these mechanisms can be integrated with virtual platforms at different levels of abstraction, resulting in a design flow that enables the rapid design space exploration of SoC designs.**

***Keywords*-System-on-chip; High-level synthesis; HW/SW co-design; Virtual platforms**

I. Introduction

System-on-chip (SoC) designs are becoming more sophisticated as the advances of the semiconductor industry allows the use of an increasingly amount of computation resources in a wide range of applications. Low-power and small *field-programmable gate arrays* (FPGAs), for instance, are now enabling the rapid deployment of complex and fully customized SoC designs even for small-scale applications. In this scenario, the strict time-to-market requirements of most applications demands a better productivity than what is possible with current *register transfer level* (RTL) methodologies, thus leading to a growing demand for *high-level synthesis* (HLS) solutions. HLS is a (semi-)automatic process that creates cycle-accurate RTL specifications from untimed or partially timed behavioral specifications. Current tools (e.g [1], [2]) already support hardware synthesis from high-level C++ and SystemC descriptions, allowing the use of higher-level implementation techniques, such as *object-oriented programming* (OOP).

These tools can be employed to create design flows in which hardware and software can be implemented in the same level of abstraction [3], facilitating the creation of component instances in both hardware and software domains. In such scenario, SoC components must be able to communicate seamlessly whether they are implemented as software running in a processor or as dedicated hardware IPs. Most design techniques do not provide such flexibility in the sense that the software views hardware circuits mostly as passive co-processors[4], [5], [2]. This may potentially lead to major system redesigns when changes in the hardware/software partitioning are performed in the final phases of the design process.

In order to contribute to this scenario, in this paper we show how hardware and software components implemented in C++ following OOP principles can be seamlessly integrated in a HLS-capable environment. This integration is performed through a *remote method invocation* (RMI) mechanism that allows cross-domain communication between hardware and software components. Our communication infrastructure is also fully implemented using C++ and *static metaprogramming* techniques [6]. This results in mechanisms that can be directly parsed by C++-based HLS tools and software compilers, thus reducing the need of non-standardized language support.

We also describe a *network-on-chip* (NoC)-based SoC platform and show how components can be deployed either as independent hardware IPs or as software running within a NoC CPU node. Furthermore, apart from a physical platform, high-level hardware simulation models of the input components and IPs may be generated and assembled to build a *virtual platform*. Virtual platforms are present in many SoC design flows [7], [8], [9] as a faster alternative to full-fledged RTL simulations. Our proposed virtual platform supports rapid functional validation and performance evaluation of the system by employing both high-level *transaction-level models* (TLM) [10] extracted directly from the original C++ implementation, and cycle-accurate models generated during the high-level synthesis process.

The remaining of this paper is organized as follows: Section II presents the programming and RMI communication models; in Section III we describe the proposed SoC platform; Section IV presents the integration of the RMI mechanisms in our design flow; Section V describes the implementation and design space exploration of a phone call processing pipeline extracted from a PABX/VoIP application; Section VI discuss related work and Section VII closes the paper with our conclusions.

II. Programming and communication model

Communication modeling is an important aspect in the design of complex embedded systems. In the software domain, components may be objects which communicate using method invocation (considering an object-oriented approach), while in the hardware domain, components may communicate using input/output signals and specific handshaking protocols. For communication through different domains, the software must provide appropriate *hardware abstraction layers* (HAL) and *interrupt service routines* (ISR), while the hardware must be aware that it is requesting a software operation.

In this work, we have chosen to rely on a pure object-oriented programming model. An OOP model has the necessary expressiveness to describe component interactions at the application level and is becoming an established standard for both hardware and software due to the introduction of C++-based synthesis. ANSI C++ and its OOP features are extensively supported by both hardware HLS tools and software compilers, thus increasing the applicability of our approach over different design flows and tools.

When focusing on an OOP-based methodology, one must provide ways to abstract the possible hardware/software partitionings from the high-level model. To illustrate this issue, Figure 1 shows a simple system represented in an OOP model. A component can be implemented as a part of another object (e.g. *C2*) creating a hierarchical object composition. Such characteristics may hinder the direct mapping between an OOP model and a physical implementation.

As show in Figure 1, the original structure may be "disassembled" in the final physical implementation if different objects in the same class hierarchy represent components that are to be implemented in different domains. For example, *C2* is "inside" *C1* in the OO model in Figure 1, but, in a possible final implementation, *C2* could be implemented as a hardware component while *C1* could run as software in a processor (first mapping in Figure 1). Furthermore, the different communication patterns between hardware and software must also be properly abstracted. For instance, in the second mapping shown in Figure 1, *C1* can call *C2*'s methods directly, while in the first mapping a hardware/software communication mechanism must be defined.

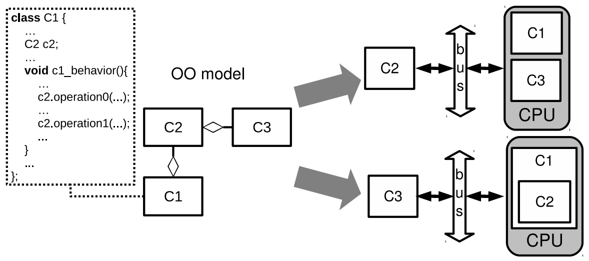

Figure 1: Possible mappings of an object-oriented model to physical implementations

To overcome this issue, we employ an approach based on RMI concepts from distributed object platforms [11]. Figure 2 illustrates the *SW–>HW–>SW* interactions that results from the first mapping shown in Figure 2. Callee components are represented in the domain of callers by *proxies*. When an operation is invoked on the components' proxy, the arguments supplied are marshaled in a request message and sent through a *communication channel* to the actual component. An *agent* receives requests, unpacks the arguments and performs local method invocations. The whole process is then repeated in the opposite direction, producing reply messages that carry eventual return arguments back to the caller.

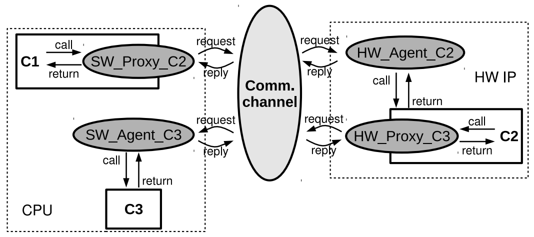

Figure 2: Cross-domain communication using proxies and agents

III. System-on-Chip platform

The implementation of *channels*, *proxies* and *agents* can be realized in several different ways (e.g. specific buses, DMA, NoCs). Figure 3 shows the chosen hardware/software architecture to implement these mechanisms. We rely on a NoC as the main communication link between hardware and software components. While a bus-based MPSoC can provide a good trade-off for software-centric designs that rely on hardware mostly as passive accelerators, it is not the most suitable choice for more heterogeneous designs in which hardware components have active roles [12], therefore in our work we have favored a NoC based architecture. The *Real-time Star NoC* (RTSNoC) [13] is the basic infrastructure of our SoC platform. RTSNoC consists of a set of *routers* with a star topology that can be arranged forming a 2-D mesh. Each router has eight bidirectional channels that can be connected to cores or to channels of other routers. Each application-specific hardware IP is deployed as a node connected to the router. Software components are compiled with a *real-time operating system* (RTOS) and run in a *CPU node* that consists of a softcore CPU and memories.

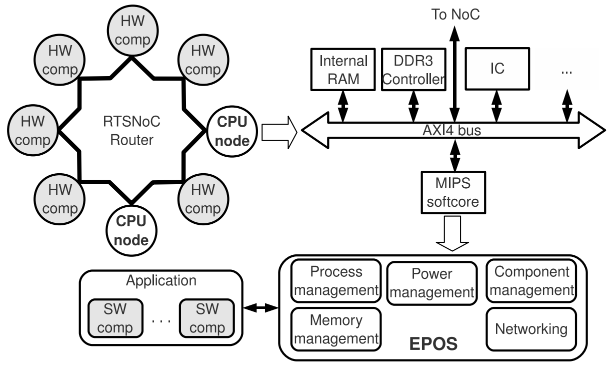

Figure 3: System-on-Chip platform

The internal structure of a CPU node is based on the AXI4 family of protocols, which is becoming the industry's standard for bus-based interconnection. In our current implementation we focused on reusability and in avoiding vendor-specific IPs, thus we have relied on IPs available at Opencores[1]. Our current CPU node, for instance, is based on the *Plasma* softcore, an implementation of the MIPS32 ISA.

The software components runs on an RTOS which provides the necessary run-time support to implement the proxies and agents described previously. For our current implementation we have chosen the *Embedded Parallel Operating System* (EPOS). EPOS is an open-source[2], multi-platform, object-oriented, component-based, operating system for embedded applications [14], [15]. Platform-independent system components implement traditional OS services, such as threads and semaphores. Hardware Mediators implement platform-specific support and are functionally equivalent to device drivers in Unix, but do not build a traditional HAL [16]. Instead, Hardware mediators are incorporated by high level components through static metaprogramming and inlining techniques. Figure 3 summarizes the main families of abstractions provided by EPOS to applications.

IV. Design flow

Our design flow consists in a set of basic steps shown in Figure 4. The developer provides high-level implementations in C++ of the software and hardware components that build up the application. An automated process then generates the respective proxies and agents which are part of the communication framework. In the software side of the flow, the components and their proxies/agents are compiled with EPOS. The hardware implementation flow encompass a process which includes: 1) C++-to-RTL synthesis using a HLS tool; 2) binding of the generated RTL descriptions and models with the pre-existing IPs to build the hardware platform; and 3) system validation using the virtual platform and its subsequent synthesis to the target FPGA device. Each of these steps is described in more details below.

[1]http://opencores.org/

[2]EPOS source code along with the sources of all components and artifacts described in this paper are available in the *EPOS project* web-page: http://epos.lisha.ufsc.br/

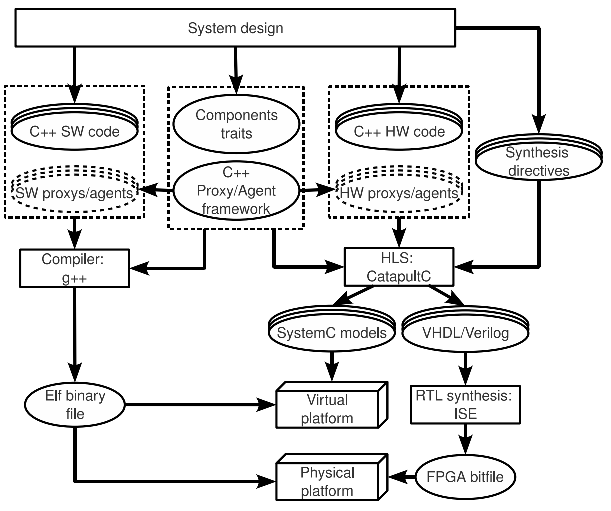

Figure 4: Design flow overview

A. Proxies and agents

The proxies and agent are automatically generated by EPOS analyzer [17]. EPOS analyzer is a syntactical analysis tool that obtains the signatures of all methods in the component's interface. With this information, a proxy or agent can be generated automatically. The example below shows the generated proxy for a software implementation of the component *C2* shown in Figure 1.

```
class SW_Proxy_C2 : SW_Proxy_Base {
  //redeclaration of the component interface
  unsigned int op0(unsigned int arg){
    //remote call implementation inherited from SW_Proxy_Base
    return SW_Proxy_Base::call_r<OP0_ID,unsigned int>(arg);
  }
};
```

The proxy only replicates the signatures of all methods in the component's public interface and implements their behavior using RMI primitives defined by the class *SW_Proxy_Base*.

The agent implementation follows a similar approach. For instance, the following is code generated for *C2*'s hardware agent:

```
class HW_Agent_C2 : HW_Agent_Base<HW_Agent_C2>, C2 {
  void dispatch(int op){
    switch(op){
    case OP0_ID:{
      // call the operation implementation from C2
      unsigned int ret = C2::op0(
            //argument deserialization implemented in base class
            HW_Agent_Base::deserialise<unsigned int>()
      );
      //return serialization implemented in base class
      HW_Agent_Base::serialize(ret);
      break;}
    default: break;
    }
  }
};
```

The mechanisms that parse the RMI messages are inherited from *HW_Agent_Base*. The component implementation is also incorporated through inheritance. Therefore, each specific generated agent must only implement a *dispatch* method which defines a switch statement that selects the correct component method using information parsed by *HW_Agent_Base*. Proxies in hardware and agents in software are also generated following the same structure.

Proxies and agents implementation

As shown above, the actual implementation of the RMI mechanisms is encapsulated in the *SW/HW_Proxy_Base* and *SW/HW_Agent_Base* classes. This allows us to separate the component-specific implementation that is part from the behavioral model from the implementation related to artifacts of the underlying platform. Extending the previous examples, the code snippet below shows part of the *SW_Proxy_Base* class, which implements an RMI request using the mechanisms provided by EPOS:

```
class SW_Proxy_Base {
  template<unsigned int OP, typename RET>
  RET call_r (){...}

  // implements a call for a one arg/one return method
  template<unsigned int OP, typename RET, typename ARG0>
  RET call_r(ARG0 &arg0){
    // serialize arguments
    char data[(sizeof(RET)>sizeof(ARG0))?sizeof(RET)?sizeof(ARG0)];
    ∗reinterpret_cast<ARG0∗>(data) = arg0;
    //RMI call
    Component_Manager::call(_type_id, _comp_id, OP, sizeof(ARG0), data);
    // deserialize arguments
    return ∗reinterpret_cast<RET∗>(data);
  }
   ...
};
```

Several template-based implementations of *call_r* are provided in order to support methods with a different number of arguments and different argument types. Each one implements the arguments serialization and requests a remote call using EPOS's *Component_Manager*. The *Component_Manager* was added to EPOS as the main software infrastructure to handle the communication between software and hardware components. It keeps lists of all existing proxies to hardware and agents to software. Each component is associated to a unique ID that is mapped by a static resource table to a physical address in the NoC. Upon a call request, this address is used to build and send packets containing the target method ID and its arguments. If the method has return values, the component manager blocks until it receives packets containing the return values.

On the hardware side, the messages are parsed by the hardware components. This parsing is implemented by the *HW_Agent_Base* class show below:

```
template<typename Component_Agent> class HW_Agent_Base {

  //agent top level interface
  #pragma hls_design top
  void top_level(ac_channel<char> &data_in, ac_channel<char> &data_out){
    int op_id = receive_call(data_in);
    static_cast<Component_Agent∗>(this)−>dispatch(op_id);
    send_return(data_out)
  }

  //serdes methods
  template<typename ARG0> void serialize(ARG0 &arg0){...};
  template<typename RET> RET deserialize(){...};

  //RMI msg handling
  int receive_call(ac_channel<char> &ch){...};
  void send_return(ac_channel<char> &ch){...};
   ...
};
```

The main responsibility of the hardware agent is to define the entry-point of the final hardware IP that is generated by the HLS tool. For the tool we have used (Calypto's CatapultC [1]), the top-level interface of the resulting hardware block (port directions and sizes) is inferred from a *single function signature*. This function is the *top_level* method and it receives an *ac_channel* object as argument. *ac_channels* are abstractions provided by CatapultC that define blocking read/write methods that can be easily coupled with the IO interface provided by RTSNoC. The behavior of the agent is to block until a packet containing a method ID is received, followed by packets containing the arguments. It parses the packets and performs the local method invocation though the *dispatch* method implemented by the component-specific implementation.

Static polymorphism is employed so that the component-specific dispatching defined at, for instance, *HW_Agent_C2*, can be called from within *HW_Agent_Base*. *HW_Agent_** classes derive from template instantiations of *HW_Agent_Base* using themselves as template. This allows the static resolution of calls to virtual methods in the base class and, therefore, yields synthesizable code.

B. Metaprogrammed framework

Another issue is how to abstract the communication mechanisms and make them transparent during component instantiation. For instance, considering the OOP example shown in Figure 1, the type `'C2'` in the `'C2 c2;'` declaration inside *C1* should be replaced by the following types according to the final hardware/software partitioning of the system:

- `HW_Proxy_C2`, if *C1* is synthesized as a hardware IP and C2 is in software;
- `SW_Proxy_C2`, if *C1* is in software and *C2* is a hardware IP;
- `C2`, if *C1* and *C2* are in the same domain. In this case, *C2* is "dissolved" inside *C1*, eliminating any communication overhead.

An efficient solution to realize these different mapping is to use *static metaprogramming* techniques [6]. Metaprograms are constructs implemented using *C++ templates* to perform operations at compile-time. An example of a metaprogram is shown below:

```
template<bool condition, typename Then, typename Else>
struct IF { typedef Then Result; };
template<typename Then, typename Else>
struct IF<false, Then, Else> { typedef Else Result; };
```

This *IF* metaprogram return one of two possible types depending on the value of a boolean condition. It is implemented using C++ partial template specialization. If *condition* is *true*, it defines *Result* as the type *Then*. This is encoded in the base definition of the template. The template is partially specialized for the *false* condition, defining *Result* as the type *Else*.

The result of a metaprogram usually depends on static configurations defined in special template classes called *Traits*. The code sample below shows two possible *Trait* classes for components *C1* and *C2*:

```
template <> struct Traits<C1> {
  static const int imp_domain = SOFTWARE;
};

template <> struct Traits<C2> {
  static const int imp_domain = HARDWARE;
};
```

They define the *imp_domain* characteristic that is used to describe in which domain the component is implemented (hardware or software).

Using the components' traits, the mapping mentioned previously for *C2* can be described using a metaprogram that replace the definition of a component by its proxy only when necessary:

```
// C2 definition
class C2 :
  public MAP<C2_Impl, HW_Proxy_C2, SW_Proxy_C2,
             Traits<C2>::imp_domain>::Result {};

//MAP metaprogram
template<typename Impl, typename HW_Proxy, typename SW_Proxy,
         bool comp_imp_domain>
struct MAP {
  typedef IF<comp_imp_domain==Traits<Sys>::curr_imp_domain,
             Impl,
           IF<Traits<Sys>::curr_imp_domain==HARDWARE,
             HW_Proxy,
             SW_Proxy>::Result> >::Result
  Result;
};
```

In the final implementation domain, *C2* can be redefined as an empty class that inherits from its actual implementation depending on the configuration defined by *C2*'s traits. In the example above, the *MAP* metaprogram selects between *C2*, *HW_Proxy_C2*, and *SW_Proxy_C2*. *MAP* uses the value of *Traits<Sys>::curr_imp_domain* to determine if the code is being submitted to a HLS tool or to a software compiler. If *Traits<Sys>::curr_imp_domain* is equal to *comp_imp_domain*, which receives the value of *Traits<C2>::imp_domain* in the example above, then it must map to the actual implementation; therefore, *Result* is equal to *C2*. Otherwise, it maps to one of the proxies according to the current implementation domain.

There exist other mechanisms to implement such mapping between implementations. Polymorphism and other dynamic language features could be used; however, such constructs are not widely supported by C++ synthesis tools. This mapping can also be implemented as part of an external tool, nevertheless the authors of this paper believe that keeping tool's dependency to a minimum and leveraging mostly on language features facilitates the migration between different C++/C-based HLS solutions.

C. Compilation/HLS

In the software flow, components and its respective proxies and agents are compiled along with the run-time support implemented in EPOS using the GCC C++ compiler. In the hardware flow, a HLS tool is used to generate an RTL IP for each component. The *traits* and the metaprograms that choose between the instantiation of an actual component or its proxy and are also part of the C++ input source code and are automatically parsed by the compiler and the synthesis tools.

In this work we have used Calypto's CatapultC [1] as the HLS tool. Additionally to the C++ source code, the inputs for the HLS process also include *synthesis directives*. Synthesis directives are used to aid the synthesis tool in associating C++ constructs to RTL microarchitectures. For instance, *loops* can have each iteration executed in a clock cycle, or can be fully unrolled in order to increase throughput at the cost of additional silicon area. The definition and fine tuning of these directives is part of the design space exploration process and is not in the scope of this paper.

D. Integration with the virtual and physical platforms

In order to provide a rapid prototyping environment using the mechanisms proposed in this work, we have developed a SystemC-based virtual platform that allows us to quickly evaluate components deployed on the hardware/software architecture described in section III. An overview of the virtual platform is shown in Figure 5. The virtual platform relies on the concept of TLM channels [10] for communication (the ▲ symbol in Figure 5). Channels are used to abstract communication details and allow components to exchange data using method calls.

In order to enable both fast functional validation and accurate performance evaluation, the virtual platform support both untimed and timed TLM models. On untimed simulation, software components are evaluated using a full MIPS32 *instruction set simulator* (ISS) that runs ELF binaries generated during the compilation process. On the hardware side, a simple TLM wrapper was developed to couple the C++ top_level defined in the Proxy/Agent framework with the TLM interface of the RTSNoC model.

On timed simulation, the untimed models are annotated with SystemC *sc::wait* statements to emulate the delays of the RTL models. For the hardware generated from C++ code, we have leveraged on the cycle-accurate models generated by CatapulC. These models are integrated with the rest of the platform through a wrapper that only drives the cycle-accurate signals between TLM transactions. This eliminates any simulation overhead from idle components. By using

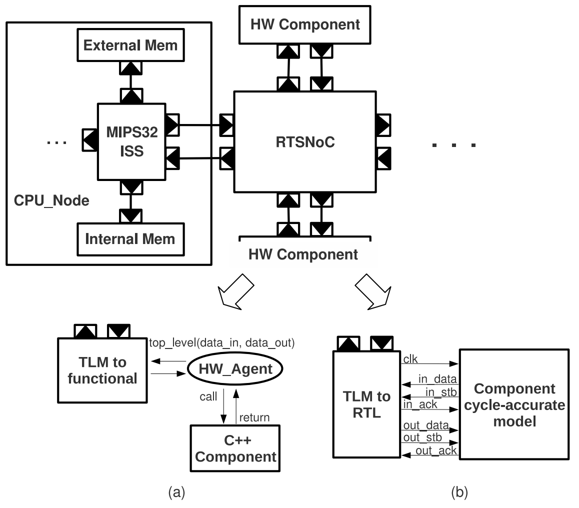

Figure 5: Virtual platform. Integration with high-level C++ components (a) and cycle-accurate models (b)

the annotated models and the models generated by CatapultC, the simulation time is significantly increased, however it is possible to obtain accurate performance data without the need of costly RTL simulations.

Once the SystemC has been verified using the virtual platform, the RTL descriptions generated by CatapultC are bound with the rest of the hardware platform library and fed to the RTL synthesis tool which generates the final hardware for the target FPGA device.

V. Case study

In order to evaluate our approach, we have also implemented part a digital PABX/VoIP system using the proposed mechanisms. This kind of system usually consists of a commutation matrix that switches connections amongst different input/output data channels. These channels are connected to phone lines (through an AD/DA converter), tone generators, and tone detectors. The system must also support the transmission of phone call data through an Ethernet network. As a case study, we have implemented a phone data processing pipeline that includes operations typical from this kind of system. Figure 6 shows the evaluated components. 4-bit ADPCM phone line samples encapsulated in encrypted packets are received from the network. An AES component first decrypts the packets using a 128-bit AES algorithm. The resulting data is then decoded to 16-bit PCM samples and send to a *dual-tone multi-frequency* (DTMF) detector. The DTMF detector uses the *goertzel algorithm* to check if a sample frame contains specific frequency components. Once a tone is detected, the system controller is notified. For this particular pipeline, the Ethernet MAC has a fixed hardware implementations while the controller is implemented only in software. For the DTMF detector, the ADPCM codec and the AES decipher we have provided both hardware and software C++ implementations. As shown in Figure 6, the components are implemented using composition as described in the previous sections.

A. Results

The goal of this experimental evaluation is to show how we can explore the different partitionings for the implemented components. By only changing the system's *traits*, the metaprogrammed framework automatically replaces the component definition by a proxy or agent when necessary. Table I shows this design space exploration.

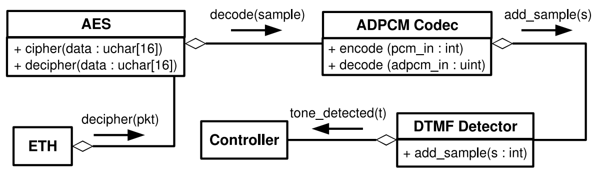

Figure 6: UML class diagram of the evaluated system. Component interactions are also indicated.

For each partitioning, columns *Virtual plat. – Untimed* were obtained by simply changing the system traits and recompiling the software and the untimed virtual platform with the high-level C++ components. As mentioned before, this does not give us any accurate performance or area figures, but enable us to quickly verify the functional correctness of different partitionings. Since the software is compiled to the target binaries at this stage, the total software footprint is already known. Columns *Virtual platform – Timed* were obtained by compiling the timed virtual platform with the cycle-accurate models generated after C++ hardware components are synthesized to RTL. At this stage we already have an initial estimation of the application execution time.The *Area rating* column also gives an initial estimation of the circuit size. Its value has no direct relation to physical units, but it is useful to compare different designs. Columns *Physical platform* shows the performance and area of the system running in the physical platform and also the simulation time of the physical platform being simulated in a full RTL simulator. The *Avg. FPGA area* is the arithmetic mean of the amount of each specific resource (e.g LUTs, flip-flops, BRAMs, MAC slices) weighted by its total amount available. This resulting value estimates the total amount of FPGA area (in %) required and is used in this paper as a unified area comparison metric.

In our experimental setup, the virtual platforms were compiled with *gcc 4.3.3* and SystemC library version 2.2. The RTL simulation was performed using *ModelSim 6.5c*. All simulations were executed in an i686 system (Intel Core2 Quad CPU Q9550 @ 2.83GHz) running Linux kernel 2.6.28 with maximum priority. Both the physical and simulated platforms were clocked at 100MHz. The software running on the MIPS32 ISS was compiled with *gcc 4.0.2* using *level 2* optimizations. For the hardware flow, *Calypto's CatapultC UV 2011a* was used to obtain RTL descriptions of the components. The descriptions were then synthesized using *Xilinx's ISE 13.4* targeting a *Virtex6 XC6VLX240T* FPGA. CatapultC and ISE were configured to minimize circuit area considering a target operating frequency of 100 MHz.

As we can see in the results, the simulation time increases by 2-3 orders of magnitude when we move to a lower level of abstraction. On the other hand, the average difference between the execution times on the real and timed virtual platform is only 13%, confirming the feasibility of our virtual platform for early and fast performance estimation. Applications with tight real-time constraint may not harness simulations that lack timing accuracy, however. We are currently aiming at improving this accuracy for future works. The difference between the execution times of the virtual and real platforms comes mostly from the MIPS32 ISS which can still be significantly improved in order to closely match the timing behavior of the Plasma softcore used in the RTL implementation. Additionally, as future work, we aim at implementing a full functional model of EPOS in order to replace the untimed ISS in the highest-level of simulation. This would allow a more significant speed-up since both hardware and software could run as host-compiled models.

The results on Table I also show that the execution time dominates

Table I: Design space exploration results. Software footprint is given in bytes. All simulation and execution times are in milliseconds.

Partitioning			Virtual plat. – Untimed		Virtual platform – Timed			Physical platform		
DTMF	ADPCM	AES	SW foot.	Sim. time	Exec. time	Area rating	Sim. time	Exec. time	Avg. FPGA area	Sim. time
SW	SW	SW	10820	2.16×10^2	26.77	274.80	5.04×10^4	33.33	0.02%	1.37×10^7
SW	SW	HW	7896	1.68×10^1	8.42	259991.93	4.39×10^3	9.90	0.90%	1.13×10^6
SW	HW	SW	10592	2.36×10^2	34.67	2090.49	6.53×10^4	38.52	0.23%	1.89×10^7
SW	HW	HW	6828	2.75×10^1	9.76	261097.72	1.88×10^4	10.73	1.05%	5.33×10^6
HW	SW	SW	8008	2.02×10^2	27.14	3733.59	5.11×10^4	31.93	0.32%	1.39×10^7
HW	SW	HW	5084	4.96×10^0	1.70	263450.72	3.27×10^3	1.79	1.20%	8.85×10^5
HW	HW	SW	6940	7.39×10^1	25.14	4839.38	4.84×10^4	27.94	0.47%	1.31×10^7
HW	HW	HW	3176	1.18×10^{-1}	0.04	263846.61	7.70×10^1	0.05	1.28%	1.99×10^4

DSE decisions for our particular case study. The fastest partitioning is about $770X$ faster than the slower, while the most efficient is terms of area is only $64X$ smaller than the least efficient. *HW/HW/HW* would be the best choice. The worst partitioning would be *SW/HW/SW* which is also the slower. AES is most demanding algorithm in the pipeline, and in this partitioning it runs in software. Additionally, all communication is performed through cross-domain RMI, which imposes a significant overhead. The *HW/HW/HW* partitioning, on the other hand, uses RMI only for the *tone_detected* method call. The rest of the pipeline is fully mapped to hardware in the same NoC node, therefore, as described in the previous sections, all communication happens through direct method calls.

VI. Related work

Several previous works have proposed mechanism for providing a uniform interface between hardware and software. The ReconOS [18] and the BORPH operating system [4] use a similar approach. In these works a task performed in hardware is seen with the same semantics as a software thread, and a system call interface is provided to hardware components. However, these works still focus on RTL as the base methodology of hardware designs. A similar approach aiming at system-level can be seen in the scope of the FOSFOR project [19], in which an infrastructure for hardware/software task has been implemented to support a future design flow based on synchronous data-flow models.

The work of *Rincón et al* [20] and *Paulin et al* [21] are based on the same concepts we have used in our approach. The authors, however, focus on RTL designs for hardware and do not provide support for designing all components at the same level of abstraction. High-level UML models of the system are only used in the former to automatically generate the communication glue necessary so that hardware and software components can communicate.

In the scope of HLS-based flows, the OSSS+R methodology uses timed SystemC descriptions for both hardware and software and defines the concept of *shared objects* [7] for high-level communication. SystemCoDesigner [8] is a tool which integrates HLS with design space exploration and provides a fully automated design flow from specification to the final platform. The design entry of SystemCoDesigner is an actor-based data flow model implemented using an extension of SystemC. The System-on-Chip Environment (SCE) [9] also defines a complete design flow. It takes SpecC models as input and provides a refinement-based methodology. Guided by the designer, the SCE automatically generates a set of TLM platforms that are further refined to pin- and cycle-accurate system implementations.

In the industry side, a considerable effort has been made by FPGA suppliers to integrate HLS solutions to their commercial design flows. Xilinx has recently incorporated C++/SystemC synthesis to its Vivado tool flow [2], providing a complete path from C++/SystemC to their standard bus-based platform. In this sense, it is worth mentioning that most of the aforementioned works also rely on Xilinx's tools. For instance, [18], [4], [20], [7], [8] provide some degree of integration with Xilinx's tools and its platform-based design flow. In our work, however, we chose to keep our proposed mechanisms as independent as possible from specific tools and methodologies.

VII. Conclusion

In this paper we have proposed a flexible NoC-based platform for SoC deployment in FPGAs and demonstrated how hardware and software components can be integrated seamlessly in a HLS-based implementation flow. This integration is done through RMI mechanisms that provide transparent communication across hardware/software boundaries. The main contribution of this work in on showing how such mechanisms can be implemented as a C++ metaprogrammed framework that integrates seamlessly with hardware/software components implemented in C++ using an object-oriented approach. Furthermore, throughout our experimental evaluation, we have shown that the integration of these mechanisms with both untimed and timed virtual platforms enables fast design space exploration and system prototyping. With our current virtual platform implementation, we have achieved a speed-up of about $260X$, with a timing accuracy of about 13% when compared to a RTL simulation. As future work, we aim at improving the virtual platforms performance and accuracy, and further explore the integration of our approach with different HLS tools and platform architectures.

References

[1] Calypto Design Systems, "CatapultC Synthesis," 2011, http://www.calypto.com/.

[2] Xilinx, "Vivado Design Suite," 2013, http://www.xilinx.com/products/design-tools/vivado/index.htm.

[3] T. R. Mück and A. A. Fröhlich, "On AOP techniques for C++-based HW/SW component implementation," in *Proc. of the 19th IEEE Int. Conf. on Electronics, Circuits, and Systems*, Seville, Spain, Dec. 2012, pp. 536–539.

[4] H. K.-H. So and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 14:1–14:28, January 2008.

[5] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473 –491, april 2011.

[6] K. Czarnecki and U. W. Eisenecker, *Generative programming: methods, tools, and applications*. New York, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[7] K. Grüttner, H. Kleen, F. Oppenheimer, A. Rettberg, and W. Nebel, "Towards a synthesis semantics for systemC channels," in *Proc. of the 8th IEEE/ACM/IFIP Int. Conf. on Hardware/software Codesign and System Synthesis*, 2010, pp. 163–172.

[8] J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, "SystemCoDesigner—an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, pp. 1:1–1:23, January 2009.

[9] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. D. Gajski, "System-on-chip environment: a SpecC-based framework for heterogeneous MPSoC design," *EURASIP J. Embedded Syst.*, vol. 2008, pp. 5:1–5:13, January 2008.

[10] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Proc. of the 1st IEEE/ACM/IFIP Int Conf. on Hardware/Software Codesign and System synthesis*, Newport Beach, USA, 2003, pp. 19–24.

[11] K. Ostrowski, K. Birman, D. Dolev, and J. H. Ahnn, "Programming with Live Distributed Objects," in *Proc. of the 22nd Eur. Conf. on Object-Oriented Programming*, 2008, pp. 463–489.

[12] G. De Micheli, C. Seiculescu, S. Murali, L. Benini, F. Angiolini, and A. Pullini, "Networks on Chips: from research to products," in *Proc. of the 47th Design Automation Conference*, 2010, pp. 300–305.

[13] M. D. Berejuck, "Dynamic Reconfiguration Support for FPGA-based Real-time Systems," Federal University of Santa Catarina, Florianópolis, Brazil, Tech. Rep., 2011, PhD qualifying report.

[14] A. A. Fröhlich, *Application-Oriented Operating Systems*, ser. GMD Research Series. Sankt Augustin, Germany: GMD - Forschungszentrum Informationstechnik, Aug. 2001, no. 17.

[15] The EPOS Project, "Embedded Parallel Operating System," 2013, http://epos.lisha.ufsc.br/.

[16] F. V. Polpeta and A. A. Fröhlich, "On the Automatic Generation of SoC-based Embedded Systems," in *Proc. of the 10th IEEE Int. Conf. on Emerging Technologies and Factory Automation*, Catania, Italy, Sep 2005.

[17] A. Schulter, R. Cancian, M. R. Stemmer, and A. A. M. Fröhlich, "A Tool for Supporting and Automating the Development of Component-based Embedded Systems," *Journal of Object Technology*, vol. 6, no. 9, pp. 399–416, Oct 2007.

[18] E. Lubbers and M. Platzner, "A portable abstraction layer for hardware threads," in *Proc. of the 2008 Int. Conf. on Field Programmable Logic and Applications*, sept. 2008, pp. 17 –22.

[19] L. Gantel, A. Khiar, B. Miramond, A. Benkhelifa, F. Lemonnier, and L. Kessal, "Dataflow programming model for reconfigurable computing," in *Proc. of the 6th Int. Workshop on Reconfigurable Communication-centric Systems-on-Chip*, june 2011, pp. 1 –8.

[20] F. Rincón, J. Barba, F. Moya, F. J. Villanueva, D. Villa, J. Dondo, and J. C. López, "Unified Inter-Communication Architecture for Systems-on-Chip," in *Proc. of the 18th IEEE/IFIP Int. Workshop on Rapid System Prototyping*, May 2007, pp. 17 –26.

[21] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, O. Benny, D. Lyonnard, B. Lavigueur, and D. Lo, "Distributed object models for multi-processor SoC's, with application to low-power multimedia wireless systems," in *Proc. of the Conf. on Design, automation and test in Europe*, 2006, pp. 482–487.