A SoC platform supporting high-level synthesis for FPGAs

Authors omitted for blind review

ABSTRACT

With the gradual increase in the complexity of *System-on-Chip* (SoC) designs, 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 flexible FPGA-based SoC platform for the deployment of SoCs in a HLS-capable environment. The proposed hardware/software infrastructure relies on a *Network-on-Chip*-based architecture to provide transparent communication mechanisms for hardware and software components. We verify the performance and area footprint of our communication infrastructure through the implementation of a SoC for digital PABX systems.

Categories and Subject Descriptors

B.6.3 [**LOGIC DESIGN**]: Design Aids—*Automatic synthesis*

General Terms

Design, Languages

Keywords

FPGA, System-on-chip, System-level design, High-level synthesis, HW/SW co-design

1. INTRODUCTION

System-on-Chip (SoC) design 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) now reaching the market are one of the products of these advances, 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.

Also known as *electronic system-level* (ESL) synthesis, HLS is a (semi-)automatic process that creates cycle-accurate RTL specifications from untimed or partially timed behavioral specifications. Current *electronic design automation* (EDA) [1, 2] tools already support hardware synthesis from high-level C++ and SystemC descriptions, enabling the use of higher-level implementation techniques, such as *object-oriented programming* (OOP). Several works have already demonstrated the applicability of HLS for implementing computational-intensive hardware accelerators [3] and the use of HLS solutions in FPGA design flows [4].

Such design flows consist in a set of basic steps. The developer provides high-level implementations (usually in C++-based languages) of the software and hardware components that build up the application. In the software side of the flow, the components are compiled and usually linked with a run-time support system, such as a *real-time operating system* (RTOS). The product of the software flow is a set of binaries for one or more *instruction-set architectures* (ISA) (e.g. in a *Multiprocessor SoC* (MPSoC) with heterogeneous processors). The hardware implementation flow is more complex and encompass a process which includes: 1) C++-to-RTL synthesis using a HLS tool; 2) binding of the generated RTL descriptions with a set of existing *intellectual properties* (IPs) (e.g. soft-processors, IO devices, memories, interconnection) in order to build a hardware platform; and 3) synthesis of the final platform to the target FPGA device.

Apart from the 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 [5, 6, 7] as a faster alternative to full-fledged RTL simulations [8]. Virtual platforms are usually composed of *instruction set simulators* (ISS) and models for the hardware components at various levels of abstraction, from *transaction-level models* (TLM) [9] to cycle-accurate models.

In order to contribute to this scenario, in this paper we describe our approach to deal with various aspects of this generic implementation flow. Our main goal is to provide a flexible hardware/software platform for the deployment of MPSoCs in a HLS-capable environment. In such flexible platform, the SoC components must communicate seamlessly whether they are implemented as software running in a processor or as dedicated hardware IP. Most design techniques do not provide such flexibility in the sense that the software views hardware circuits only as passive co-processors. This may potentially lead to a major system redesign when changes in the hardware/software partitioning are performed in the final phases of the design process. A uniform communication infrastructure is a requirement for design strategies in which the same component can have different implementations in different domains (hardware or software) [10]. In order to deal with these issues, we have employed concepts from distributed object platforms [11] to implement cross-domain communication using a remote call mechanism. A uniform component communication framework is also provided for both hardware and software. The framework is implemented using *static metaprogramming* techniques [12], resulting in mechanisms that can be easily used across different hardware platform and HLS tools. Our proposed platform was designed in order to provide the necessary hardware/software infrastructure to support the framework implementation. We rely on a *Network-on-Chip* (NoC) interconnect in order to provide a scalable and flexible design.

The remaining of this paper is organized as follows: Section 2 presents a discussion about works related to software/hardware communication and HLS flows targeting FPGAs; Section 3 presents the component communication framework; in Section 4 we describe the proposed SoC platform along with the evaluation of the communication mechanisms; Section 5 extends our experimental results and describes the implementation of a *Private Automatic Branch Exchange* (PABX) SoC using our platform; Section 6 closes the paper with our conclusions.

2. RELATED WORK

Several previous works have proposed mechanism for providing a uniform interface between hardware and software in FPGA-based platforms. The *HThread* project [13], the ReconOS [14] and the BORPH operating system [15] 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 [16], 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* [11] is based on the same concepts we have used in our approach and also proposed a NoC-based architecture. The authors focus on RTL designs for hardware. However, they rely on high-level UML models of the system to automatically generate all the communication glue necessary so that hardware and software components can communicate seamlessly.

In the scope of HLS-based flow targeting FPGA, the OSSS+R methodology [5] uses timed SystemC descriptions for both hardware and software and defines the concept of *shared objects* [17] for high-level communication. SystemCoDesigner [6] 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) [7] 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 that are further refined to pin- and cycle-accurate system implementation.

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 acquired AutoESL [2] and a complete path from C++/SystemC to their standard bus-based platform is already available [4]. In this sense, it is worth mentioning that most of the aforementioned works also rely on Xilinx's tools. For instance, [14, 15, 11, 17, 6] provide some degree of integration with Xilinx's EDK 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. Also, we have favored a NoC based architecture over the bus-based one generated by Xilinx's flow. While a bus-based MPSoC provides 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 [18].

3. COMPONENT COMMUNICATION FRAMEWORK

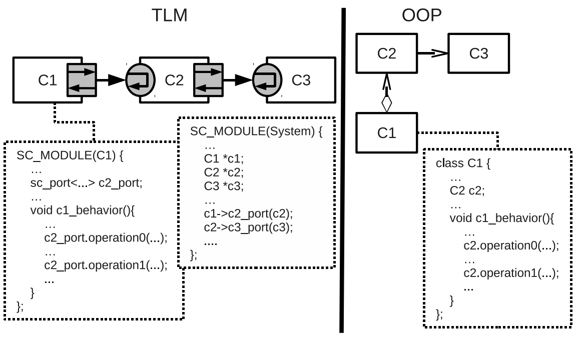
Communication modeling is an important aspect in the design of complex embedded systems. Table 1 summarizes the most common components communication patterns. In the software domain, components may be objects which communicate using method invocation (considering an object-oriented approach), while in 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.

Table 1: Usual component communication patterns. The *Caller* requests operations from the *Callee*.

Direction	Type of communication	
	Caller	Callee
SW→HW	HAL sends commands to the HW using a communication infrastructure (e.g. a bus or a NoC).	Communication interface receives commands and triggers the operations.
HW→SW	HW interrupts SW and waits for the operation to finish. It may transfer data to/from the main memory (e.g. DMA).	An ISR calls the requested operation and signalizes HW when finished.
SW→SW	Function call interface	
HW→HW	Signals/Handshaking	

Before providing an abstraction for these mechanisms, the basic programming model must be defined. Since in this work we aim at providing a generic mechanism for communication between both hardware and software, we have chosen to rely on a pure object-oriented programming (OOP) model. An OOP model has the necessary expressiveness to describe component interactions at the application level and is a becoming an established standard for hardware and software due to the introduction of C++-based synthesis.

Figure 1 shows a simple system represented in an OOP model. For illustrative reasons, the same system is also defined using a *transaction level modeling* (TLM) [9] approach. TLM, and its underlying language SystemC [19], is another approach for modeling at the system level. TLM rely on the concept of *channels* for communication. Channels can be used to abstract communication details and allows components to exchange data using calls. Latter in the design process, these channels can be mapped to buses, point-to-point signals, or function calls, whether the communicating components are in hardware or in software. This strategy provides a clear separation between communication and behavior, but do not provide the same expressiveness of the OOP model. In the OOP representation shown in Figure 1, for instance, a component can be either implemented as a global object (*C1* and *C3*) or as a part of another object (*C2*). Another important motivation to rely on OOP is that ANSI C++ and its OOP features are extensively supported by hardware HLS tools. Implementations based on C++/OOP can be directly used as input to a software/hardware compiler, thus increasing the applicability of our approach over different design flows and tools.



```
SC_MODULE(C1) {
    ...
    sc_port<...> c2_port;
    ...
    void c1_behavior(){
        ...
        c2_port.operation0(...);
        ...
        c2_port.operation1(...);
        ...
    }
};
```

```
SC_MODULE(System) {
    ...
    C1 *c1;
    C2 *c2;
    C3 *c3;
    ...
    c1->c2_port(c2);
    c2->c3_port(c3);
    ....
};
```

```
class C1 {
    ...
    C2 c2;
    ...
    void c1_behavior(){
        ...
        c2.operation0(...);
        ...
        c2.operation1(...);
        ...
    }
};
```

Figure 1: Communication between components in transaction level (left) and pure object-oriented (right) models.

Still, the structure of TLM-like models naturally leads to more coarse-grained components whose structure is closer to the final physical implementation. In OOP, the original structure will end up "disassembled" if different objects in the same class hierarchy represent components that are to be implemented in different domains. For instance, *C2* is an attribute "inside" *C1* in the OO model in Figure 1, however, in a possible partitioning, *C1* could be implemented as a hardware component while *C2* could run as software in a processor.

To overcome this issue, we employ an approach based on concepts from distributed object platforms [11]. Figure 2 illustrates possible interactions between components *C1* and *C2*. The callee is represented in the domain of the caller by a *proxy*. 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.

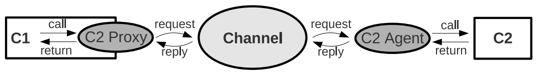


Figure 2: Cross-domain communication using a proxy and an agent.

3.1 Metaprogrammed framework

The implementation of *channels*, *proxies* and *agents* can be realized in several different ways and depends on the underlying computer architecture (e.g. using buses, DMA, NoCs) and RTOS features. To deal with this variability, we have defined a hardware/software communication framework that supports different implementations of proxies and agents. The framework is shown in Figure 3.

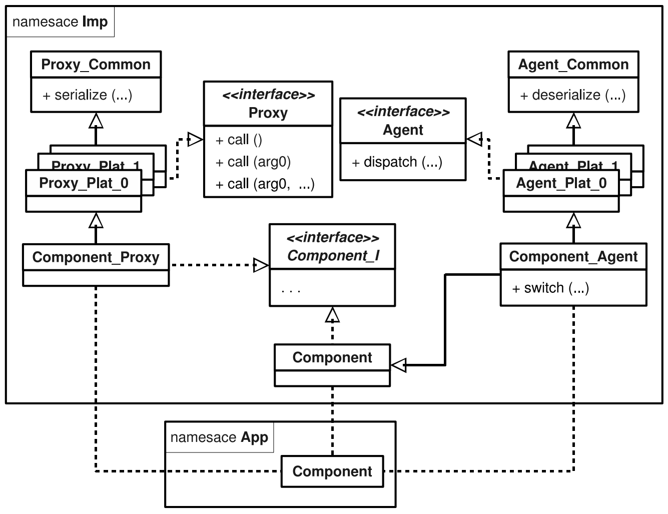


Figure 3: UML class diagram of the communication framework

Platform-independent behavior (e.g. the marshalling algorithm) is encapsulated in the common classes *Proxy_Common* and *Agent_Common*. These classes are then extended to provide platform specific implementations. These platform-specific implementations are further extended to obtain the proxies/agents for each component. The implementation of component-specific proxies is straightforward and can be automatically generated by analyzing the method call interface of each component. For instance, in Figure 3, *Component_Proxy* must only implement the same public interface of *Component* using the marshalling methods provided by its base class. In the opposite side, the *Component_Agent*'s *switch* method must only define a switch statement that selects the correct local method using information parsed by its base-class (e.g. method id, number of arguments, and argument's data).

The framework shown in Figure 3 is the same for hardware and software, but, for each class, a specific implementation must be provided for each domain according to the target architecture and HLS tools. For instance, the *dispatch* method of an agent in hardware must define the entry-point of the component according to the requirements of a specific HLS tool, allowing the generation of an IP with the expected IO interface. More details about the specific implementation of these mechanisms are provided in Section 4.1.

Another issue is how to make these mechanisms transparent during component instantiation. Figure 3 shows that the same component is defined twice in different namespaces: *Imp::Component* and *App::Component*. The definition inside the *Imp* namespace is the actual implementation of the component, while the definition inside *App* is the one used to instantiate the component and must map to the correct definition inside *Imp*. For instance, considering the OOP example shown in Figure 1, the declaration of *C2* inside *C1* can be defined as *App::C2*, which may have the following different mappings according to the final hardware/software partitioning of the system:

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

Additionally, *App::C2* maps to an agent when it is the current component being compiled/synthesized and it receives methods invocation from other components in different domains. An efficient solution to realize these different mapping is to use *static metaprogramming* techniques [12]. Metaprograms are constructs implemented using *C++ templates* to perform operations at compile-time. The result of a metaprogram 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 bool hardware = true;
};

template <> struct Traits<C2> {
  static const bool hardware = false;
};
```

It defines a *hardware* characteristic that is used to define in which domain the component is implemented (hardware or software). Using the components' traits, the mapping mentioned above for *C2* can be described in the hardware domain using the following metaprogram:

```
namespace App {
typedef HW_MAP<Imp::C2, Imp::C2_HW_Agent, Imp::C2_HW_Proxy,
               false,
               Traits<C2>::hardware>::Result
        C2;
}
```

HW_MAP selects between *Imp::C2*, *Imp::C2_HW_Agent*, and *Imp::C2_HW_Proxy* according to the values of the last two parameters. The implementation of the *HW_MAP* metaprogram is shown below:

```
template<class Component, class Agent, class Proxy,
         bool top_level, bool hardware>
struct HW_MAP {
  typedef IF<top_level,
             Agent,
           IF<hardware,
             Component,
           Proxy
    >::Result>::Result Result;
};

//IF metaprogram
template<bool condition, class Then, class Else>
struct IF { typedef Then Result; };

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

The value of *top_level* is *true* when the component is the one being currently synthesized and a top-level interface is required, thus a mapping to an agent. In the example, the synthesis scope is defined by *C1*, thus the value *top_level* for *C2* is *false*. If *C2* is also in hardware, then its own implementation is returned. In this case, *C2* is incorporated by *C1* during the synthesis process. If *Traits<C2>::hardware* is *false*, then *C1* will have access to *C2* through its proxy.

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 EDA 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 tools and flows.

4. SYSTEM-ON-CHIP PLATFORM

The mechanisms proposed in the previous sections are mostly general guidelines that can span several specific implementations. The implementation of *channels*, *proxies* and *agents* can be realized in several different ways (e.g. specific buses, DMA, NoCs). Figure 4 shows the general structure of the chosen hardware/software architecture to implement these mechanisms. We rely on a NoC as the main communication link between hardware and software components. The *Real-time Star Network-on-Chip* (RTSNoC) [20] is the core component of our SoC platform. RTSNoC consists in 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 IP generated from a hardware component is deployed as a node connected to the router(*App node*). Software components are compiled with an RTOS and run in a *CPU node* that consists of a softcore CPU and memories. Shared IO peripherals are encapsulated in separated *IO nodes* so they can be used by both hardware and software components.

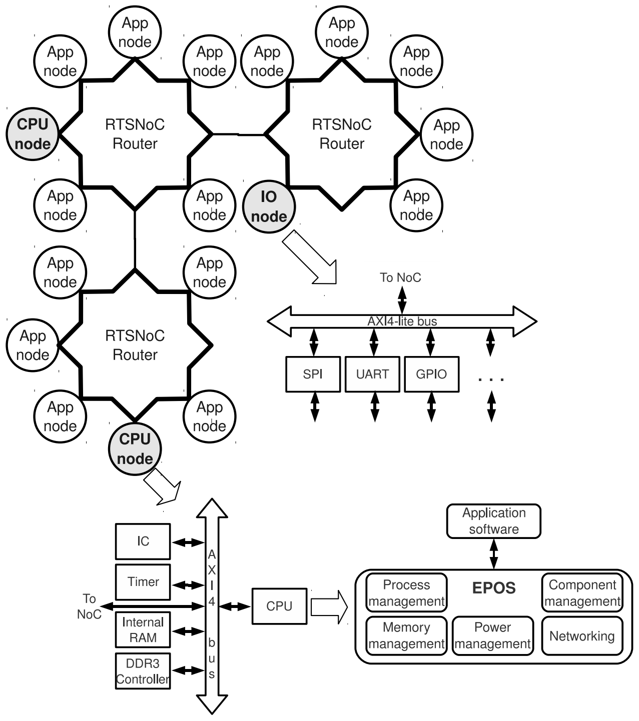


Figure 4: System-on-Chip platform

The internal structure of CPU and IO nodes is based on the AXI4 [21] 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 [22]. 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) [23]. EPOS is an open-source lightweight operating system for dedicated applications and provides the necessary run-time mechanisms. Additionally, it supports multi-core architectures and reconfigurable platforms with dynamic reconfiguration. Figure 4 summarizes the main families of abstractions provided by EPOS to applications. In the next section we describe the mechanisms related to component communication.

4.1 Proxies and agents implementation

As defined in Figure 3, a common marshalling process is implemented in the *Proxy_Common* and *Agent_Common* classes for both hardware and software. The support for RTSNoC and EPOS is performed in platform-specific classes that inherit the marshalling from the common classes. These are further extended to perform the binding between each component method call interface and the proxy/agent. In order to illustrate the relationship between the different elements of our platform, Figure 5 shows the interaction between the implemented proxies and agents during cross-domain method calls.

The interactions in Figure 5 are based on the example shown in Figure 1 assuming the following partitioning and behavior:

- *C1* is in software, *C2* is in hardware, and *C3* is in software;
- *C1* calls the method *C2::op0*. Inside *C2::op0*, *C2* calls *C3::op1*;
- both methods have one argument and one return value

A call to *C2::op0* is converted by *C2_HW_Proxy* (using methods provided by the base class) to a remote call request handled by the *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 packets containing the target method ID and its arguments. These packets are sent through the NoC using the class *NoC_Wrapper*. If the method has return values, the component manager blocks until it receives the return values from the hardware component.

For hardware components, the dispatcher defined by the agent is also used to set 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 *dispatch* 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. Figure 5 show the behavior of the dispatcher defined by *C2_HW_Agent*. It blocks 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 in *C2*. *C2* then calls *C3* through its proxy. Proxies in hardware are very similar to proxies in software, except by the fact that there is not a global component manager. All the required mechanisms are implemented locally in each proxy. Since C3's proxy is in the same node of C2 and its agent, they both share the same *ac_channel*.

Packets sent to the CPU node trigger interrupts that are handled by an ISR defined in the component manager. The ISR reads all pending packets and performs the necessary operations. When the manager receives a packet containing return values, it searches a list of blocked proxies and forwards the packet to the correct one (in Figure 5 the *call* method issued by *C2_SW_Proxy* returns with the return value). When a packet contains data from a method call request (e.g. packets 2 and 3 in Figure 5), the information is forwarded to the dispatcher of the respective agent. The *dispatch* method of a software agent is very similar to the

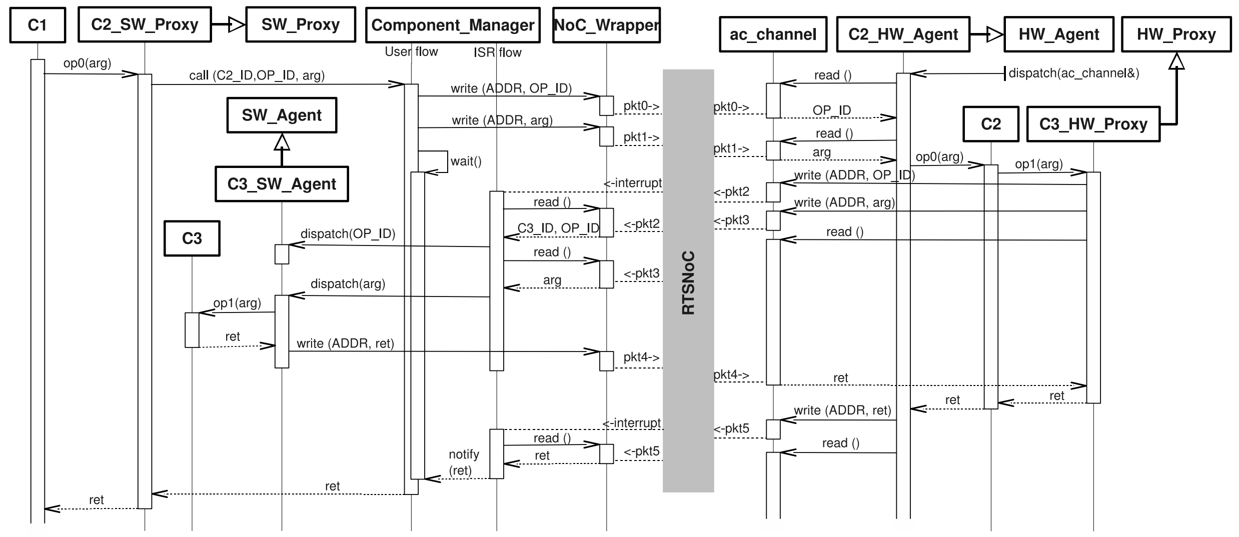


Figure 5: UML sequence diagram showing the interaction between proxies and agents in SW->HW->SW calls

hardware implementation, but it receives data through successive explicit calls. Once all the data is received, the local call is performed in the software component.

4.2 Implementation flow

So far we have not integrated our approach in a fully automated design flow. Nevertheless, the mechanisms based on standard C++ solutions allow for a straightforward integration with current synthesis tools. To illustrate this, Figure 6 shows the steps we are currently using to go from C++ descriptions to components deployed in the platform. These steps are described below:

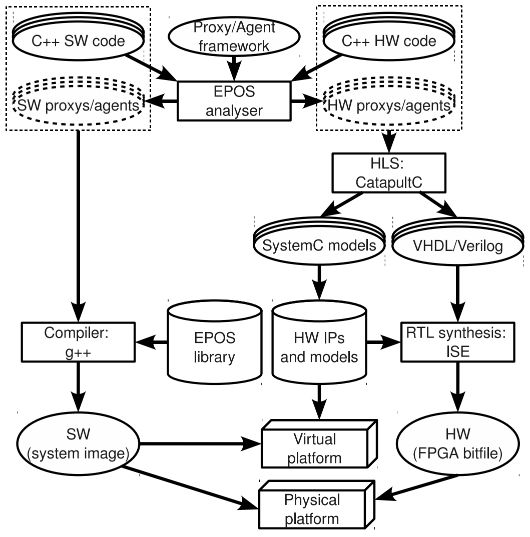


Figure 6: Steps for obtaining RTL hardware descriptions and cycle-accurate models from C++ code

4.2.1 *Proxies and agents generation*

Due to the modular hierarchy of the communication framework, the generation of proxies and agents for each component can be easily automated once a generic implementation is provided for the platform. The RTOS used in this work also provides a toolchain that includes a syntactical analyzer that can be used to obtain the operation signatures of all components [24]. The code sample below shows the generated code for *C2* from the previous example:

```
SW_AGENT_BEGIN(C2,1)
    SWITCH_BEGIN
        CALL_R_1(op0, OP0_ID, unsigned int)
    SWITCH_END
SW_AGENT_END

SW_PROXY_BEGIN(C2,C2_ID)
    unsigned int op0(unsigned int arg){
        return call_r<OP0_ID,unsigned int>(arg);
    }
SW_PROXY_END
```

Preprocessor macros are used to shorten the syntax of classes' definitions and to assure a uniform naming convention if the manual implementation of a proxy or agent is required. The macros *SW_PROXY_BEGIN* and *SW_PROXY_END*, for instance, simply define the inheritance shown in Figure 3. They are defined as follows:

```
#define SW_PROXY(name) name##_SW_Proxy
#define SW_PROXY_BEGIN(name,id)\
class SW_PROXY(name) : private SW_Proxy<Imp::name,id> {\
public:\
    typedef SW_Proxy<Imp::name> Base;\
\
    SW_PROXY(name)() :Base() {}\

#define SW_PROXY_END };
```

4.2.2 *Hardware synthesis*

In this work we have used Calypto's CatapultC [1] as the HLS tool. The inputs for the HLS process consist of C++ code and *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 output of the HLS process are RTL descriptions in VHDL or Verilog and SystemC models at various levels of abstractions for design verification.

4.2.3 Platform generation and simulation

The RTL descriptions are bound with the hardware platform library and fed to the RTL synthesis tool which generates the final hardware for the target device. The platform library contains the implementation of the core components of the SoC: CPU/IO nodes and the RTSNoC. We have leveraged on the cycle-accurate models generated by CatapulC to provide a *virtual platform* for the complete system. We have implemented a full ISS for the MIPS32 ISA and timed TLM models of the RTL IPs available in the platform library. These models are integrated with the ones generated by CatapulC to provide a fast and accurate virtual platform that can be used to verify the performance and behavior of the final system without the need of costly RTL simulations.

4.3 Performance evaluation

In order to provide an overview of our platform's performance, we have first measured the run-time of some applications running on the basic SoC components (a CPU node and an IO node). For this initial evaluation we have selected the following set of benchmarks from *MiBench* [25]: **Dijkstra** calculates the shortest path between nodes using Dijkstra's algorithm; **SHA** executes a secure hash algorithm; **FFT** calculates a 4096-point floating point complex FFT; and **Susan** runs a set of recognition algorithms over a 76x95 raw image. Table 2 show the execution time of the benchmarks running on both the virtual and physical platforms, as well as the total simulation times of both platforms. The benchmarks were compiled with *gcc 4.0.2* targeting the MIPS32 ISA and using *level 2* optimizations, while the virtual platform was compiled with *gcc 4.3.3* and SystemC library version 2.2. The RTL simulation was performed using *ModelSim 6.5c* [26]. 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. As can be seen in Table 2, the average difference between the execution time on the real and virtual platforms is only 11%, while the improvements in terms of simulation time were over 275 times, 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. Even so, we believe it is still feasible to cover this kind of applications with our virtual platform. 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 behavior performance of the Plasma softcore used in the RTL implementation.

We have also evaluated the performance of our communication infrastructure in terms of latency and resource utilization. Tables 3 and 4 show the total area footprint of proxies/agents for both hardware and software. For the hardware structure, the RTL descriptions were synthesized using *Xilinx's ISE 13.4* targeting a *Virtex6 XC6VLX240T* FPGA. CatapultC and ISE were configured to minimize circuit area

Table 2: Virtual platform accuracy and simulation time. *S-Corners* and *S-Edges* stands for Susan's corner and edge detection algorithms, respectively.

Benchmark	Exec. time (ms)		Simulation time	
	V. plat.	Real plat.	V. plat.(s)	RTL(min)
SHA	33.31	36.21	6.9	32.1
Dijkstra	2074.86	2290.38	555.3	2493.3
FFT	5180.96	6042.62	1350.4	6231.9
S-Corners	42.38	47.96	8.9	42.2
S-Edges	54.59	61.90	11.5	52.1

considering a target operating frequency of 100 MHz. All implementations achieved similar clock frequencies and met the 100 MHz constraint.

Table 3: FPGA resource utilization of proxies/agents and the main platform components

	Proxy	Agent	RTSNoC	Nodes	
				Proc.	IO
6-input LUTs	45(10)	61(16)	2256	7775	537
Flip-flops	89(2)	147(4)	689	6834	409
36Kb BRAM	0	0	0	2	0
Avg. area	0.02%	0.03%	0.30%	1.90%	0.11%

Table 4: Memory footprint of proxies/agents and the run-time support

	Proxy	Agent	Comp. manager	Others
Code	64	364	2422	9756
Data	4	44	74	381
Total	68(46)	408(136)	2496	10137

In both tables, proxies and agents support a single method with a 4-byte argument, while the *()'s* values indicates the average amount of extra resources required to support an extra method invocation. For instance, a hardware agent for a component with two methods in its interface requires about 80 (61 + 16) LUTs to be implemented. The *average area* shown in Table 3 is the arithmetic mean of the amount of each specific resource weighted by its total amount available. This resulting value estimates the total amount of FPGA area (in %) required and can be used as a unified area estimation metric.

To determine the significance of the area values, Table 5 compares our results with data extracted directly from previous works. These works [11, 27] are described in more detail in section 2 and provide a proxy/agent infrastructure similar to our proposal. On the hardware comparison, the difference is significant and can be explained by the simple interface required by RTSNoC in relation to the complex IPIF protocol used by [11]. On the software side, the achieved memory footprint is about 10% smaller than that of the reference work [27]. Taking into account that the latter targeted an 8-bit architecture that requires less memory to store instructions, we can conclude that our component management infrastructure yields a low overhead in terms of memory footprint.

Table 5: Full area footprint of hardware/software proxies for a component defining two methods with one parameter and one return value. For software, this area includes the Component Manager.

Work	Plat. type	HW agent (LUTs/FFs)	SW proxy (code+data)
Ours	32-bits MIPS / RTSNoC	80 / 154	2636
[11]	32-bits PPC / IPIF bus	202 / 354	-
[27]	8-bits AVR	-	2911

Figure 7 shows the number of cycles required for a method invocation between software and hardware in both directions (SW->HW and HW->SW). As reference, we also show the "no overhead case" in which both components are implemented in the same domain and callee invocation are not inlined in the caller's code. In the HW->HW communication, this refers to an interaction between components implemented in different NoC nodes.

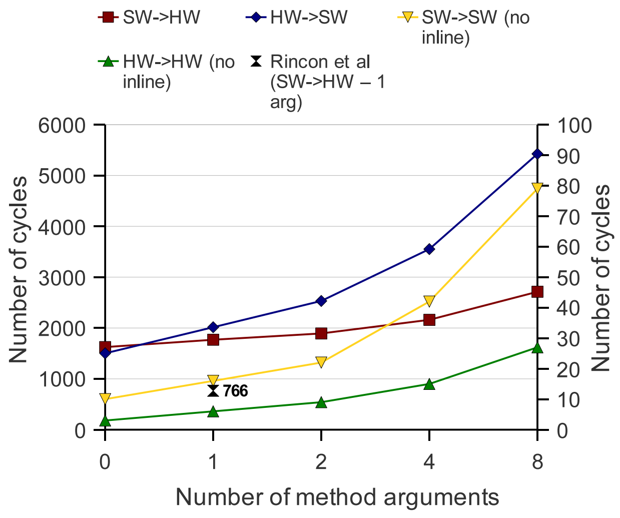


Figure 7: Latency of a method invocation between different implementation domains. The SW->SW and HW->HW communication are aligned with the rightmost Y-axis.

For SW->HW, a method invocation with a single 4-byte argument takes 1769 cycles, while the each additional argument increases the latency by about 135 cycles. In the opposite case, the same invocation takes 2015 cycles and each additional argument adds about 500 cycles. Figure 7 also show the latency of a similar invocation in the same related work compared in Table 5 [11]. Our apparent higher latency can be explained by the performance of the Plasma softcore used in our current implementation, which is considerably less powerful then the PPC processor used in the related work.

5. CASE STUDY

In order to further evaluate our approach, we have also implemented a PABX application using the proposed platform. A high-level diagram of the PABX system and its mapping to the platform is shown in Figure 8. The main component is 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 also supports the transmission of voice data through an Ethernet network.

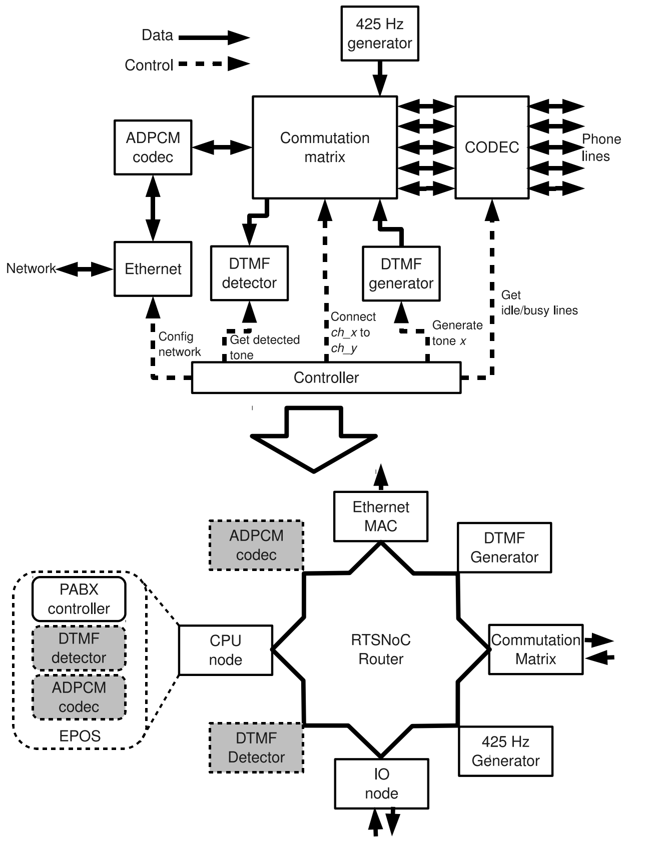


Figure 8: PABX system diagram and mapping to the SoC platform

In order to analyze the impact of our transparent communication mechanisms, we have selected the following key components of the application and provided C++ implementations for both hardware and software: an 16-bit IMA ADPCM encoder/decoder [28] that is used to reduce the traffic of data transmitted through the network; and a *Dual-Tone Multi-Frequency* (DTMF) detector that uses the *goertzel algorithm* to check if a sample frame contains specific frequency components. These components are highlighted in Figure 8 and appear in as hardware and software, since they can move between both domains depending on the final configuration of their *Traits*.

5.1 Results

Table 6 shows the communication overhead of cross-domain method calls. For the ADPCM codec, each call transmits a sample and returns its enc/decoded counterpart. Since the algorithm itself is not computationally intensive, the time to transmit the arguments and receive the return values dominates the total time of the operations. The DTMF detector provides the opposite case. Requesting a tone detection (*do_dtmf* method) requires less cycles than a tone detection. However, the total running time of this component is completely dominated by communication overhead if cross-domain calls are used to update the detector's sample buffer (*add_sample* method). Nevertheless, in practice this overhead is eliminated when the DTMF detector is in hardware. Samples are only sent by the commutation matrix which is always in hardware, while detections are only requested by the PABX controller which is in software.

Table 6: Total communication overhead (OH) of cross-domain method calls

Component		Execution time (cycles)		
		Behavior	Comm.	OH
HW->	*encode*	404	2015	83.3%
ADPCM(SW)	*decode*	324	2015	86.1%
HW->	*add_sample*(x700)	47027	1051400	95.7%
DTMF(SW)	*do_dtmf*	540803	1502	0.3%
SW->	*encode*	1	1769	99.9%
ADPCM(HW)	*decode*	1	1769	99.9%
SW->	*add_sample*(x700)	700	1136800	99.9%
DTMF(HW)	*do_dtmf*	6341	1624	20.4%

Figure 9 shows the total memory footprint and FPGA area of components supporting a remote call. The data was obtained by compiling and synthesizing each component along with its agent. Due to optimizations performed by the toolchain (e.g. function inlining), it is not possible to identify the agent overhead in the final object code and netlists, so the overhead highlighted in Figure 9 is calculated based on the data from Tables 3 and 4. The calculated overhead varies from 11.1% (HW DTMF) to 33.7%(SW ADPCM) of the total component area.

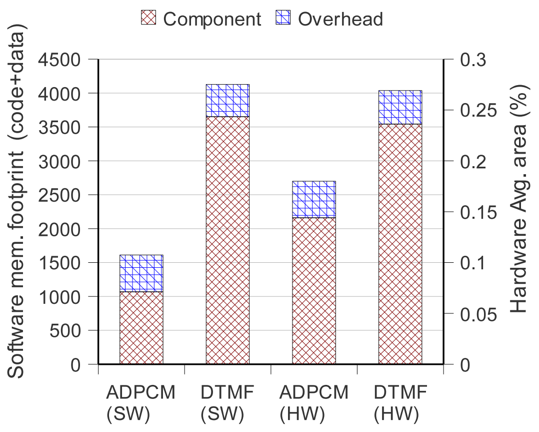


Figure 9: Memory footprint and FPGA area

Table 7 shows the total footprint considering two different partitioning of the case studies: both components in hardware, and both in software. To highlight the overhead added by the component management mechanisms, the footprint *()'s* value of a specific partitioning is the total footprint subtracted by the ones of all case studies. By comparing the *()'s* values, we can see that, in the *All HW* partitioning, the introduction of the component management mechanism in software added an overhead of 3638 bytes On the hardware side, the calculated area overhead when all cases are in hardware represents only 1.3% of the total circuit area and negligible when represented in terms of the total amount of resources available in the target device (only 0.07%).

Table 7: SoC area footprint

	Memory (code+data)	FPGA Avg. rsc. util.
All SW	19553 b (14689 b)	3.70% (3.70%)
All HW	18327 b (18327 b)	4.25% (3.77%)

6. 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 possible through the use of *proxies* and *agents*. These artifacts implement a remote call mechanism that allows the transparent communication across hardware/software boundaries. A metaprogrammed framework isolates these concepts from the behavior of the system. The effort of migrating a software component to hardware, or vice-versa, is reduced to coding the core functionality in the target domain, therefore avoiding systemwide redesigns and facilitating late hardware/software partitioning. Furthermore, our mechanisms are implemented using only standard C++ features, thus increasing compatibility with different C++/C-based HLS tools and implementation flows.

Finally, we have shown experimental measurement on cross-domain call overheads and communication performance and have demonstrated the application of our approach on a real-world case study. Compared to previous works, our mechanisms yield a small overhead in terms of memory and FPGA resources. The impact on performance is considerable; however, this is due to the low performance of the CPU core used in our current implementation. We are currently working to extend our platform with higher performance CPU cores, which will significantly reduce the execution time of software routines. Throughout our experimental evaluation, we have also identified another important aspect which is the subject of ongoing and future work: establishing component communication through method calls only favors control-oriented applications. New features will be considered in order to provide a better support for high throughput data-oriented applications. Our main concern with this aspect is on the seamless integration of streaming interfaces and DMA mechanisms with our object-oriented modeling approach.

7. REFERENCES

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

[2] Xilinx, "AutoESL High-Level Synthesis," 2012, http://www.xilinx.com/tools/autoesl.htm.

[3] M. Fingeroff, *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010.

[4] 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.

[5] A. Schallenberg, W. Nebel, A. Herrholz, P. A. Hartmann, and F. Oppenheimer, "OSSS+R: a framework for application level modelling and synthesis of reconfigurable systems," in *Proc. of the Conference on Design, Automation and Test in Europe*, Nice, France, 2009, pp. 970–975.

[6] 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.

[7] D. Rainer, G. Andreas, P. Junyu, S. Dongwan, C. Lukai, Y. Haobo, A. Samar, D. Daniel *et al.*, "System-on-chip environment: a SpecC-based framework for heterogeneous MPSoC design," *EURASIP Journal on Embedded Systems*, vol. 2008, 2008.

[8] M.-C. Chiang, T.-C. Yeh, and G.-F. Tseng, "A QEMU and SystemC-Based Cycle-Accurate ISS for Performance Estimation on SoC Development," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 593 –606, april 2011.

[9] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Proc. of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Newport Beach, USA, 2003, pp. 19–24.

[10] H. Marcondes and A. A. Fröhlich, "A Hybrid Hardware and Software Component Architecture for Embedded System Design," in *Proc. of the International Embedded Systems Symposium*, Langenargen, Germany, 2009, pp. 259–270.

[11] F. Rincón, J. Barba, F. Moya, F. J. Villanueva, D. Villa, J. Dondo, and J. C. López, "Transparent IP Cores Integration Based on the Distributed Object Paradigm," in *Intelligent Technical Systems*, ser. Lecture Notes in Electrical Engineering, 2009, vol. 38, pp. 131–144.

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

[13] E. Anderson, W. Peck, J. Stevens, J. Agron, F. Baijot, S. Warn, and D. Andrews, "Supporting High Level Language Semantics within Hardware Resident Threads," in *Proc. of the International Conference on Field Programmable Logic and Applications.*, Aug. 2007, pp. 98 –103.

[14] E. Lubbers and M. Platzner, "A portable abstraction layer for hardware threads," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, sept. 2008, pp. 17 –22.

[15] 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.

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

[17] 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 international conference on Hardware/software codesign and system synthesis*, ser. CODES/ISSS '10. New York, NY, USA: ACM, 2010, pp. 163–172.

[18] 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*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 300–305. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837352

[19] P. R. Panda, "SystemC: a modeling platform supporting multiple design abstractions," in *Proc. of the 14th international symposium on Systems synthesis*, Montr&#233;al, Canada, 2001, pp. 75–80.

[20] 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.

[21] IEEE, "AMBA AXI Protocol Specification (Rev 2.0)," 2010, http://www.arm.com.

[22] "Opencores," April 2011, http://opencores.org/.

[23] The EPOS Project, "Embedded Parallel Operating System," 2011, http://epos.lisha.ufsc.br/.

[24] 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.

[25] "Mibench suite," April 2011, http://www.eecs.umich.edu/mibench/.

[26] Mentor Graphics, "ModelSim," 2011, http://model.com/.

[27] G. Gracioli and A. A. Fröhlich, "ELUS: A dynamic software reconfiguration infrastructure for embedded systems," in *Proc. of the IEEE 17th International Conference on Telecommunications*, april 2010, pp. 981 –988.

[28] Interactive Multimedia Association (IMA), *Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems*, 1992. [Online]. Available: http://www.cs.columbia.edu/h̃gs/audio/dvi/IMA_ADPCM.pdf