Object-Based Communication Architecture for System-on-Chip Design

Jesús Barba, Fernando Rincón, Francisco Moya, Juan Carlos López, Julio Dondo
Dept. of Technology and Information Systems
School of Computer Science, University of Castilla-La Mancha
Ciudad Real, Spain
Jesus.Barba@uclm.es

Abstract—**In this work, we present an integrated approach to the SoC design problem based on a mixed (HW and SW) implementation of a system-level middleware specifically designed for SoCs: the Object-Oriented Communication Engine (OOCE). OOCE provides a high-level and homogeneous view of the SoC components based on the Distributed Object paradigm. The resulting communication infrastructure easies the integration of the HW and SW parts, allows the automatic generation of the HW/SW interfacing adapters and also enables true concurrent design methodologies.**

To prove the viability and efficiency of our proposal a prototype implementation on the Xilinx-V2 Pro platform has been developed.

Keywords-SoC design; automatic synthesis; object-oriented design methodology; HW/SW interfacing

I. Introduction

Currently, the concept of SoC (*System-on-a-Chip*) is the maximum exponent of the continuous improvement in the manufacturing process of integrated circuits. However, important shortcomings arise in traditional development methodologies because of the complexity of such kind of system. Hence, getting a set of hardware and software pieces and making them work together almost without effort is a utopia.

Heterogeneity in communication infrastructures, IPs (*Intellectual Property*) cores coming from different providers and the dependency of the hardware platform to start coding the embedded software are some of the challenges a SoC designer must face.

The magic recipe, which has been widely adopted by the academia and CAD industry, to cope with the above mentioned problems, consists in raising the abstraction level of the specifications that will drive the design and implementation processes of a SoC. The final objective is to shorten the time invested in the development process by means of reusing existing components and systems, automating the more tedious and error-prone tasks and parallelizing the work of the hardware and software teams [1].

Therefore, the election of the right system model that would effectively enable the above mentioned design techniques must be in the core of any successful proposal.

In this paper, we come up with a comprehensive solution to recurrent problems in SoC design: *the Object-Oriented Communication Engine* (OOCE from now on). OOCE makes several contributions to the state-of-the-art in embedded system design from different points of view:

- *Hardware*. OOCE easies component integration and promotes the reuse of legacy IPs.
- *Software*. OOCE offers a unified and high-level communication interface for both HW and SW components which simplifies the programming task.
- *Methodology*. OOCE becomes as an enabler for concurrent HW/SW design. OOCE also automates the generation of the solution.

OOCE puts the focus on the definition of the interface components and communication models of a SoC. It provides a set of services on top of a component model that can be used to implement several models of computation. Thus, designers only have to use such facilities after modeling the application to implement the system, avoiding an important amount of work that they probably perform ad-hoc for every application.

The chosen component and communication model for SoCs is based on the *distributed object paradigm*. The reasons for such selection can be summarize in: (1) capability to unify HW/SW communication interface, (2) suitability for reconfigurable computing due to object built-in state management and serialization mechanism, and (3) transparent management of the communications.

A. Related work

Lately, the middleware concept has attracted the interest of the embedded system community as a feasible solution to many of the problems sketched in the previous section.

For example, Paulin et al. in [2][3] present Multiflex which is a CORBA-inspired middleware infrastructure for embedded systems, mainly targeted to multimedia and wireless applications. Mutiflex also relies on the distributed object paradigm to offer an homogeneous view of the interface communication between HW and SW. However, there are no clear references in Multiflex regarding how transparency (key concept in any middleware) is managed in it.

This research was supported by the Spanish Ministry of Science and Innovation under the project DAMA (TEC2008-06553/TEC), and by the Regional Government of Castilla-La Mancha under project RGRID (PAI08-0234-8083).

Many other works borrow well known concepts from the middleware microcosm but not define a completely communication infrastructure such as Multiflex or OOCE. In this line, it is worth mentioning the work of Klingauf et al. It describes how the concept of *Hardware Procedure Call* (HPC) [4], on top of *Transaction Level Modeling* concepts, offers a truly high-level access mechanism to HW functions in a service oriented manner. There are no references to the architecture of the resulting hardware/software supporting platform for HPCs and its efficiency. HPC can be considered as an HW implementation of the *Remote Procedure Call* semantics, the basis of many non-object based middlewares.

II. The Proposal

OOCE is a system-level middleware for SoCs based on a distributed object model and the *Remote Method Invocation* (RMI) semantic (present in current state-of-the-art software middlewares) to seamless integrate the HW and the SW parts of a SoC. The middleware is an abstraction layer that provides advanced communication services independently of the underlying hardware platform, programming language or operating system used.

OOCE presents a hybrid communication infrastructure specifically tailored for SoCs where many of the entities that conforms the platform are implemented in HW so that the maximum efficiency with the minimum overhead can be guaranteed. On top of this, a set of tools have been developed to allow the automatic generation of most of the infrastructure from a system high-level specification. OOCE provides a set of basic services upon which new ones with a higher degree of complexity can be built.

Throughout the rest of this paper we will give more details of the OOCE architecture, the services that have been defined and the tools that allow building OOCE-based implementations using a semi-automatic design flow.

III. A Distributed Object Model for SoCs

The Distributed Object Model (DOM) provides the components in a bus-based SoC with the necessary semantics for intercommunication. By semantics we mean how such components represent the data to be exchanged, and how they use the bus interface signals to assure a correct communication and data delivery.

The DOM defined by OOCE, which is specifically adapted for SoCs, includes: (1) a recommendation of implementation of objects as hardware modules, and (2) a specification of how invocations between objects within a SoC are mapped to *read* and *write* transactions over the interconnection infrastructure (i.e. the RMI protocol for SoCs).

A. OOCE Hardware Objects

The concept of hardware object is used to reduce the gap between system specification and the final implementation. The aim of OOCE is not to constrain how an object in the model must be implemented, but to recommend a common way to interact with the cores to help to automate the generation of OOCE communication adapters (*proxies* and *skeletons*). Otherwise, the designer should write the necessary glue logic to adapt the different interfaces as in traditional design flows. A hardware object comprises: (1) a standardized and simple interface to model point-to-point connections with the IPs that implement the object behaviour, (2) a *local* (point-to-point) method invocation protocol, and (3) mechanisms to retrieve and set the state of the object.

One of the main features of the hardware object model is the flexibility to define how method parameters are written/read to/from the IP (synchronous or asynchronous local invocations) as well as the size of the data ports. This makes it easier to fit the final implementation to particular design constraints and also to adapt existing IPs.

B. OOCE RMI Protocol

The OOCE RMI protocol defines the number and type of messages that the client object exchanges with the server object to invoke a method in the second. The signature of a method determines the type of invocation to be performed: *one-way* or *two-way*. One-way invocations are only possible when there are neither outputs nor return values in the method definition. In this case, a sole *request message* flows from the client to the server. When a two-way invocation takes place, the client expects, after the execution of the method, a *response message* from the server with the results. OOCE can manage the invocation process both asynchronously or synchronously.

The OOCE RMI protocol also defines the format of the message to be sent and the encoding rules for the message payload. The format of the messages remains unchanged, no matter the nature of the communicating objects. This means that a target object is not able to distinguish whether the source of the invocation is a SW or a HW object. This is essential to offer access and location transparency.

Finally, message delivery consists in a sequence of *write* operations on the destination until the message body, is completely sent. The target address of a message is derived from its header fields. Since *write* operations are basic services offered by any bus technology, the complete process can be easily targeted to any particular one. If available, advanced communication mechanisms such as bursting or split transactions may be used.

IV. OOCE in Detail

The OOCE architecture is shown in Fig. 1, where the supported communication scenarios are also depicted. Proxies and skeletons (the middleware communication adapters) are the most important components since they are on the basis of the RMI semantics. Any method invocation takes place between a proxy and a skeleton, which translate the OOCE RMI protocol to the particularities of the in-chip communication channel. OOCE provides the tools for the automatic generation of such components, so developers do not have to deal with the cumbersome task of designing and implementing such adapters.

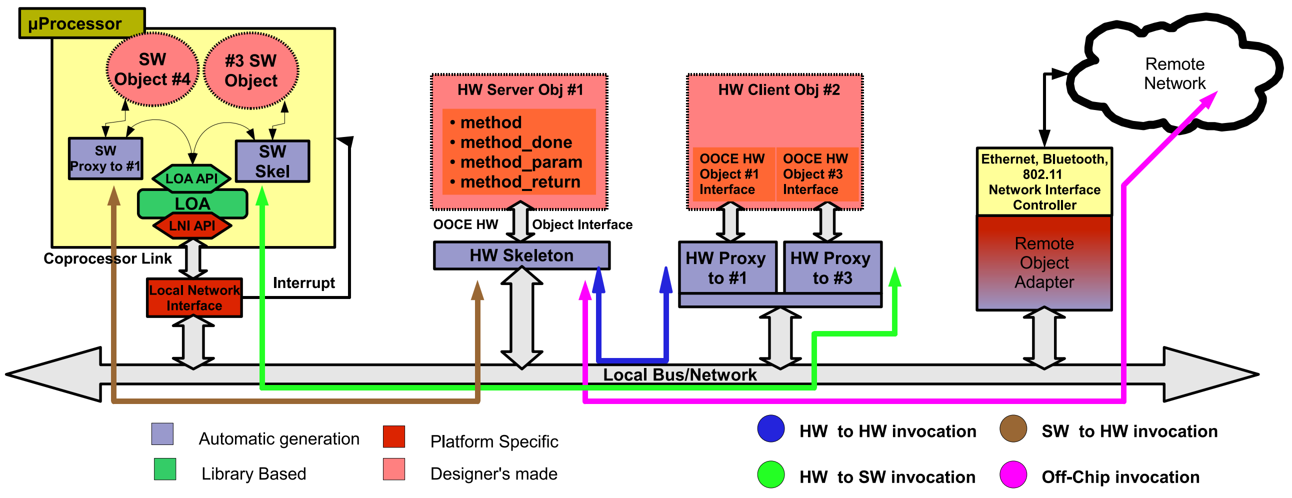


Figure 1. Main components of the OOCE architecture.

The *Local Object Adapter* (LOA) and the *Local Network Interface* (LNI) manage the communications between HW and SW components. The LOA is platform independent and it is provided as part of the OOCE component library. On the other hand, the LNI (conceived as a coprocessor) depends on the interface with the processor, so some parts must be written ad-hoc for every new target platform.

The *Remote Object Adapter* (ROA) provides connectivity with external objects and/or systems. One side of the ROA depends on the component that acts as the bridge with the external network so it has to be hand made. However, on the side interacting with the local on-chip network, the control logic is fully customizable and generated in an automatic way.

Following, the different communication scenarios and services are analyzed.

A. HW to HW invocation

HW proxies and skeletons isolate the physical components that implement the client and the server HW objects from the bus. They are also in charge of adapting the point-to-point invocation protocol to the OOCE RMI semantics. Several templates for HW proxies and skeletons have been defined, which combine synchronous-asynchronous and direct-indirect invocation styles. These templates are specialized (in an automatic way, using an interface compiler) according to the signature of the methods offered, so the resulting implementation is optimal.

A layered approach for the architecture definition of these elements has been followed. In the highest level, the adaptation of the data according to the local method invocation mechanism imposed by the HW object interface is done. In the lowest one, bus specific signalling is carried out in order to *deliver* the message built in upper layers according to the remote protocol rules.

Skeletons and proxies are able to perform re-ordering of requests and responses if required. This allow to implement a *concurrency transparency* mechanism in an asynchronous communication scenario as well as a *quality of service* mechanism, delaying the processing or delivering of protocol messages according to an encoded priority parameter.

B. HW/SW communication

HW/SW communication is also transparently handled in OOCE. Therefore neither HW nor SW objects must change the way they interface depending on which the target/source of a method invocation is. The RMI protocol assures that any method invocation involving HW and SW objects uses exactly the same messages than the ones generated in a HW to HW method call. Due to this, the HW templates for proxies and skeletons need no modifications to support a HW-to-SW or SW-to-HW communication. This keeps the OOCE HW/SW interfacing infrastructure to a minimum.

The Local Network Interface is the bridge between the system microprocessor, where the SW application runs, and the HW cores. The main goal of the LNI is to keep the HW interface and the activation protocol defined in OOCE compatible with the SW invocation mechanisms. The LNI routes the relevant bus traffic to a SW object, which is running in the processor. To abstract the link with the system processor, we have defined a layered software architecture that progressively offers services that help the programmers to transparently use this communication infrastructure.

C. Off-chip communication

OOCE provides transparent off-chip communication with external components implementing the ICE protocol. ICE is an object-oriented commercial middleware widely used in the industry. The Remote Object Adapter (ROA) is the OOCE component responsible for offering such functionality. Existing on-chip objects can communicate with both, external SW servers written using ICE and external HW servers implemented as OOCE components. Also, on-chip OOCE objects can be accessible from outside. It is worth to spot that previously existing in-chip components do not have to suffer changes in order to fit with the new communication scenario.

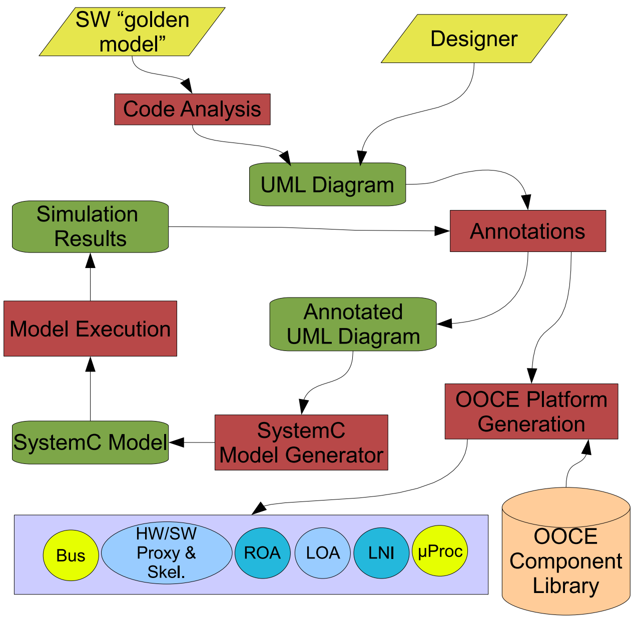


Figure 2. Simplified design flow in OOCE.

V. Advanced Features

A set of advanced services and applications have been built upon the basic communication facilities. These services can be used to ease the development of end-user applications or more complex services. Some of these services are an efficient HW implementation of solutions that are usually handled by the OS, leading to a considerable performance improvement:

- *Location service (LS)*. The LS provides methods to lookup and update a table of references. This table relates a physical reference with a logical reference to an object. The LS is mainly oriented to provide indirect communication (IC) in a SoC.
- *Group Communication (GC*). The broadcast nature of bus-based communications within a SoC can be logically incorporated into the DOM to provide group communication. A special set of logical references to objects in the system are reserved and identified as *group identification references*. The implementation of a exception (errors) communication subsystem and a service discovery protocol (applied to reconfigurable objects) are the two GC main applications.
- *Synchronization component library (SCL).* We have developed a HW version of mutexes, semaphores and mailboxes to easily adapt pre-existing concurrent applications.
- *Reconfiguration Service (RS).* Details of a RS for adaptive, dynamic applications (using dynamic re-configurable logic) based on OOCE can be seen in [5].
- *Run-time failure management.* The LS may have several physical references to a set of objects that implement the same functionality. If an error is detected in one of them, IC mechanism will get a valid reference from this pool of objects. The replacement object can be indistinctly implemented in SW or in HW.

VI. SoC Development in OOCE

To offer a complete support to SoC design based on OOCE, we have developed a design methodology (see Fig. 2) based on the model, micro-architecture and services previously presented.

All the scripts and interface compilers are able to automatically generate: (1) the OOCE adapters (both HW and SW), (2) the platform, through the selection and specialization of the communication engine elements required by the application from a component template library, and (4) the Xilinx EDK project files to start working with.

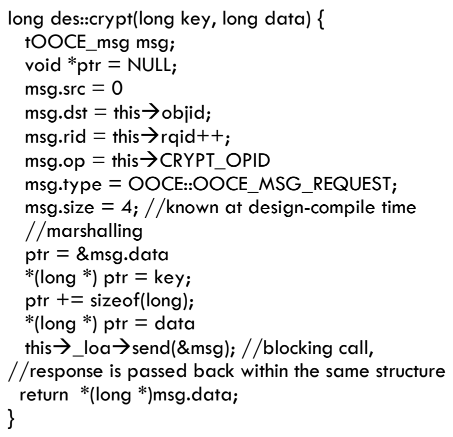
The entries to this final phase in the design framework are two sets of UML diagrams. The target platform is specified using the modeling capabilities of the OMG profile for MARTE whereas the application is modeled by means of standard UML *object* and *collaboration* diagrams. Such diagrams are annotated by the stereotypes defined in an OOCE UML profile in order to specify (among many other aspects): (1) whether an object is going to be a SW or a HW object, (2) the processor where the SW object is going to run, (3) a concrete scheme of communication (synchronous or asynchronous), or (4) the bus infrastructure and protocols used to integrate the components by means of a association with the corresponding platform component diagram.

We propose an iterative design framework that, starting from a software-only object model of the system, progressively reaches a heterogeneous implementation. This is possible thanks to the access and location transparency principles supported by OOCE.

The SystemC interface compiler generates the models for the proxies and skeletons as well as all the SW/HW interfacing OOCE components. Such model is intended to be used for a rapid assessment of the overhead introduced by the chosen configuration. The simulation gives data about the time spent in each remote operation invocation (execution + RMI protocol), the number of executions of a method, etc. Such profiling information can be used to: (1) detect potential communication bottlenecks, (2) calculate the required bandwidth, and (3) help in the deployment of the components and the election of a HW or SW implementation for each one…; in brief, to help the designer in the exploration of the design space.

A. *HW/SW Interface compilers*

In this section, we focus on HW/SW interfacing, key in the synthesis process of Hardware-dependent Software (HdS) which has become one of the principal matters of concern in embedded systems due to the increasing amount of software that such systems include [6].



```
long des::crypt(long key, long data) {
   tOOCE_msg msg;
   void *ptr = NULL;
   msg.src = 0
   msg.dst = this→objid;
   msg.rid = this→rqid++;
   msg.op = this→CRYPT_OPID
   msg.type = OOCE::OOCE_MSG_REQUEST;
   msg.size = 4; //known at design-compile time
   //marshalling
   ptr = &msg.data
   *(long *) ptr = key;
   ptr += sizeof(long);
   *(long *) ptr = data
   this→_loa→send(&msg); //blocking call,
//response is passed back within the same structure
  return  *(long *)msg.data;
}
```

Figure 4. Generated C++ code of a DES crypto SW proxy

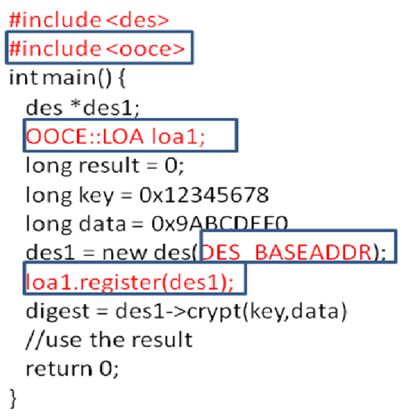
In the case of HdS, objects provide a more stable development scenario, and at the same time they set the basis of good *design for reuse* practices. In other approaches, the interfaces offered to the programmers are low-level and very sensitive to variations in the hardware cores (i.e. the access through a register bank interface). Software developers may then invest most of their time and effort in rewriting small parts of the software, leading to a more unproductive work.

In OOCE the automatic generation of the HW/SW interfacing infrastructure, boost the productivity of the embedded software developers because (1) they do not have to wait for a physical platform prototype, and (2) unnecessary iterations are avoided.

To illustrate this affirmation let us think in a simple application that may need the cryptographic services of a component which implements the DES algorithm. Fig. 4 shows a simple version of how the main program looks like before and after making the required changes to the code in order to use the OOCE middleware. As it can be seen, the implementation details of the communication with the DES core are hidden behind the OOCE HW/SW interfacing infrastructure. The most important changes are related to the instantiation of the OOCE runtime support object where the proxy to the DES core must be registered in. The DES object constructor suffers only a slight modification since it is mandatory to provide the physical base address of the IP core that implements such functionality. Nonetheless, the middleware also defines a method to obtain such address at runtime which is useful in dynamic environments.

A new version of the DES class, which is interface-compatible with the former one, is generated by the interface compilers. This class is in charge of interacting with the middleware. Fig. 3 shows some parts of the constructor and proxy to the *des* method.

The resulting code for the main application is clean and easy to understand which promotes reuse and maintainability. Also, the application is more robust to unforeseen changes in the hardware platform and almost the entire software stack can be used ‘as is’ in future designs. Therefore, the writing of the embedded software can start as soon as desired, even in parallel with the design of the hardware design. No matter the physical



```
#include <des>
#include <ooce>
int main() {
   des *des1;
   OOCE::LOA loa1;
   long result = 0;
   long key = 0x12345678
   long data = 0x9ABCDEF0
   des1 = new des(DES_BASEADDR);
   loa1.register(des1);
   digest = des1->crypt(key,data)
   //use the result
   return 0;
}
```

Figure 3. Original main program and modifications (boxed sentences).

interface of the hardware core, it can be modified or replaced by a different core from another manufacturer; the logical interface remains invariable.

VII. EXPERIMENTAL EVIDENCE

In order to evaluate the viability and efficiency of our proposal, we have compared a non-OOCE-based implementation of a simple image processing application with the OOCE-based one. Such application consists of a video capture, a space color converter and a video sink components which are connected to the system bus. The goal is to compare the resource consumption as well as the communication overhead by means of measuring the frame rate achieved. In both versions we have used the same hardware cores for the three main application components enumerated above, and the target platform is a Xilinx Virtex2Pro prototyping board.

For the first version (non-OOCE) of this application we used the tool chain provided by Xilinx in order to generate the IPIF core adapters for the PLB bus. The MicroBlaze executes a software routine which is in charge of configuring and controlling the movement of data using the DMA engine. The software uses the provided Xilinx drivers to interface with the IPs.

In version two (OOCE-based), we have generated the core bus wrappers making it use of our interface compilers. As mentioned before, the interfacing software routines were also obtained. Contrary to the non-OOCE version, the control software routine running in the processor has little responsibility regarding data movement between blocks. Such duty is now delegated to the component wrappers. Proxies and skeletons are able to synchronize and exchange data buffers on their own, without the intervention of the software. This is possible because the HW to HW asynchronous invocation semantic introduced by OOCE. The interface to software is limited to the initial configuration (setup of the producer and consumer logical links – SW to HW invocations) and the error control flow (exception notification – HW to SW invocations).

In both versions, after the core wrapper generation, it was needed some manual work in order to adapt the wrapper logic to the core interface.

TABLE I. RESOURCE AND TIMING COMPARISON

	System Version	
	Non-OOCE	***OOCE-based***
LUTs	12105	11258 (-7%)
FFs	7710	7324 (-5%)
Measured Frame Rate (fps)	13	15 (+15%)

Table I shows all the results regarding implementation efficiency and communication overhead. Our system proves to be more efficient since it does not suppose an increment in the required resources mainly due to a significant reduction in the logic for the bus adapters (up to 40% less).

Moreover, a better frame rate is also achieved by the OOCE-based implementation since most of the control operations avoid the use of software routines. This represents a reduction of the processor load since most of the control data flow traffic does not necessary pass through the processor.

Regarding implementation details of each OOCE platform components, following we present a brief summary. All of them have been prototyped on the Xilinx XUP-V2Pro board.

The OOCE component library comprises two versions of the LNI coprocessor (one for the Microblaze using the FSL interface and one for the Power PC using the DCR bus) and a customizable core for the ROA (OPB and PLB versions) that works with the Ethernet MAC core from Xilinx. Regarding the SW side, the LNI_link layer had to be tailored to the FSL and the DCR interfaces aforementioned. On the contrary, only one implementation of the LOA layer was done since it is platform independent. The total size of the software stack (excluding the generated SW proxies and skeletons) is 90 lines of C code which has a positive impact in the overhead introduced in HW/SW communication. The interface compilers are written in C++ and, currently, they only support the generation of VHDL for HW and C++ or C for SW. They both are thought to be easily extended to other implementation languages such as Verilog or Java if required.

The evaluation of the extra HW resources needed by the OOCE infrastructure is quite satisfactory. In average, each HW wrapper only represents about 1% of the total logic used by a single core. When we compare the HW proxies and skeletons with the equivalent Xilinx IPIF solution, OOCE demands much less resources (this reduction ranges from 20% to 40%). Moreover, HW proxies and skeletons together with the LNI provide exactly the same communication services to SW resources than IPIF and increment the HW/SW communication bandwidth up to 40%. This important increment in performance is due to the efficiency of the software layer and the small delay introduced by the LNI component in HW/SW invocations (just 6 cycles for incoming traffic and 3 cycles for an outgoing invocation). The LNI core represents an increment of about 5% and 3% regarding the logic used by the Microblaze and the PowerPC respectively.

The ROA core is able to process ICEP messages two orders of magnitude faster than its counterpart in SW. The ROA is able to parse a complete frame (Ethernet, TCP and ICE headers), check and validate the packet and translate the object and method identification strings to internal bus addresses in less than 90 microseconds.

VIII. CONCLUSIONS

In this paper, a complete approach for SoC design based on a distributed object model is presented. OOCE defines a light-weight, efficient communication architecture for systems that are modeled as communicating objects. The principal features of OOCE are: (1) flexibility since it is extremely easy to adapt it to new target technologies, (2) it provides the same programming interface for HW and SW elements (which boost the productivity of the embedded software developers), (3) it adds the necessary semantics to directly translate invocations to an implementation level using elemental communication services, (4) it supports advanced services to ease the management of complex tasks such as synchronization, migration, replication, etc., and (5) most of its components are generated in and automatic way.

REFERENCES

[1] Jerraya, A.A., "HW/SW Implementation from Abstract Architecture Models," Proc. DATE'07, 2007, pp. 1-2.

[2] P.G. Paulin et al. *Parallel Programming Models for a Multiprocessor SoC Platform Applied to Networking and Multimedia*, IEEE Transactions on VLSI systems, vol. 14, 17, July 2006.

[3] P.G. Paulin et al. *Distributed Object Models for Multi-Processor SoC's, with Application to Low-Power Multimedia Wireless Systems*. In Proc. of Design Automation Conference, Mar 2006.

[4] Klingauf, W. et al.., "Embedded software development on top of transaction-level models". *CODES+ISSS '07*, 2007.

[5] Dondo, J. , Rincon, F., Barba, J., et al. ,"Dynamic reconfiguration management based on a distributed object model," Proc FPL'07, 2007, pp.684-687

[6] A.A. Jerraya, A. Bouchhima, F. Pétrot, "Programming models and Hw-Sw interfaces abstraction for Multi-Processor SoC", In Proc. of the 43th Design Automation Conference, San Francisco, California, 2006.