A Comprehensive Integration Infrastructure for Embedded System Design[☆]

Jesús Barba[a,*], Fernando Rincón[a], Francisco Moya[a], Julio Daniel Dondo[a], Juan Carlos López[a]

[a]*Department of Technology and Information Systems, School of Computer Science, University of Castilla-La Mancha, Spain*

Abstract

A System-on-a-Chip (SoC) is the most successful example of how the evolution of the chip integration technology allows the manufacture of complex embedded systems. However, the bulk of the design effort, to efficiently combine the HW and SW components in a SoC, still resides in the HW/SW interfacing architecture. A good HW/SW integration strategy has a positive impact either in performance, efficiency, development times, productivity or reutilization of platforms for future designs.

In this paper, we present an object-oriented approach to cope with the HW/SW integration problem in SoCs. The Object-Oriented Communication Engine (OOCE) is a system-level middleware particularly designed for SoCs which provides a high-level and homogeneous view of the system components based on the Distributed Object paradigm. Communication between components is abstracted by means of a HW implementation of the Remote Method Invocation semantics and all the SW and HW adapters are automatically generated from functional descriptions of the components interface. The resulting communication infrastructure simplifies the integration effort required and makes the embedded software more resilient to changes in the HW platform.

To prove the viability and efficiency of our proposal a prototype imple-

[☆]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 SAND (PEII11-0227-0070).

[*]Corresponding author. Phone: +34 926295300 Ext: 3708 Fax: +34926295354

Email address: `jesus.barba@uclm.es` (Jesús Barba)

mentation on the Xilinx ML505 evaluation platform has been performed.

Keywords:
System-on-Chip co-design, interface synthesis, object-oriented design methodology for embedded systems, HW/SW interfacing

1. Introduction

Lately, the ever increasing complexity of electronics systems has presented ever greater challenges for hardware architects and embedded software engineers. The ultimate goal in SoC (*System-on-a-Chip*) design would be very close to what happens in a software engineering project where the programmer takes everything he needs from a component library, writes some lines of glue code and has it up and running almost without effort. But the development of systems and applications on embedded platforms is trickier than the ideal scenario described above. Some of the problems are stated below:

a) The heterogeneity and the large number of the components to be assembled. Some of them might not even have been developed yet, as in the case of functionality to be implemented as core hardware after a profile analysis of the target application.
b) The variety of communication infrastructures to be used and the varying nature of the communication protocols (e.g., buses or Networks-on-Chip).
c) First hardware, then software workflow [1]. Hardware dependent software is quite sensitive to changes in the physical platform, becoming a bottleneck in embedded system projects. The alternatives are to delay embedded software development or assume the risk of unforeseen changes in the platform.

The magic formula, which has been widely adopted by academia and the CAD industry, to address the above mentioned problems consists of raising the abstraction level of the specifications that will drive the design and implementation processes of a SoC. Together with a proposal for a common system model, a *path to implementation* must be provided if it is intended to be useful and attractive to designers and, of course, productive.

In this scenario, we come up with a novel and comprehensive solution to the recurrent problems present 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:

a) *Hardware.* OOCE facilitates component integration and promotes the reuse of legacy IPs (Intellectual Property), shortening the development times.
b) *Software.* OOCE offers a unified and high-level communication interface for both HW and SW components which simplifies the programming task.
c) *Methodology.* OOCE becomes an enabler for concurrent HW/SW design. OOCE also automates the generation of the solution without the need for detailed or formal specifications which means an increment of the designer's productivity.

The rest of the paper is organized as follows. First, we present the related work. In Section 3 we introduce the design philosophy behind OOCE and its advantages before describing the main guidelines that drive the realization of the Distributed Object Paradigm in SoCs (Section 4). Section 5 depicts the design flow and tools supporting the OOCE approach. In Section 6 the OOCE architecture is outlined before ending in Section 7 with the experimental results. We draw our conclusions in the last section.

2. Related Work

HW/SW interfacing has been a subject of great interest for decades under a variety of different names (e.g. *System Level Design, Co-design or Electronic System Level*). However, the problem remains a hot topic nowadays, fed by the need for new tools, methodologies and design techniques in order to deal with the market requirements.

The quest for *the most suitable model to abstract embedded platforms* has opened three main research lines: integration through Operating System (OS), object models and component models. The tag *OS-based models* groups those techniques using the task, process or thread [2, 3] concepts.

OS abstractions are present in a majority of projects concerning HW/SW interfacing. For example OS4RS [4] provides a uniform communication scheme for hardware and software tasks. However, wrapping a hardware core in a task shell is not straightforward and needs modifications in the OS internals, compilers or linkers as in IRES [5], which tend to increase HW/SW data transfer times. On top of that, hardware resources are accessible through file-system primitives, exporting a low-level interface in form of memory or register accesses [6, 7]. Although the kernel interface simplifies the development of applications (since it is familiar to programmers), it is not clear that such interface facilitates the reuse of hardware. To avoid the high latency and low interface abstraction limitations present in previous OS based approaches, Lange et al. [8] propose a fine-grained HW/SW execution model and an architecture that supports it.

Contrary to the OS based approaches, object-oriented and component based solutions are conceived as a virtualization layer usually built on top of the OS. Objects and components provide an interface of a higher level of abstraction than the previous solutions.

For example, the work of Paulin et al. in Multiflex [9] is an example of a complete middleware for embedded systems based on the distributed object-oriented paradigm. Other approaches use the same principles but aimed at offering solutions to more restrictive scenarios. Klingauf et al. present the concept of Hardware Procedure Call (HPC) [10] as a high-level mechanism to access HW functions in a service oriented manner. HPC can be considered as a HW implementation of the Remote Procedure Call semantics, the precursor of the Remote Method Invocation and basis of many non-object based middlewares.

The work of Gailliard et al. [11] defines a mapping of the CORBA (an object-oriented middleware for networked computers) semantics (Interface Definition Language and General Inter-ORB Protocol) to OCP semantics. The goal is to provide an interoperability and integration framework for hardware components. A shift from object-based to component- based can be also found in [12, 13]. In [12] a bridge component is used to abstract processors, buses, embedded OS, etc. from embedded platforms. The bridge is specified for every platform and is responsible for propagating events across the HW/SW boundary.

Lately, components have gained increasing importance in HW/SW co-design claiming to be an evolution of objects with enhanced features for a better reutilization of the building blocks [13]. However, from our point of view, there is not a favorite regarding the use of components or objects in a design. Indeed, any component model could be realized using object-oriented artifacts (i.e. *CORBA Component Model*).

3. A Distributed Object Model for SoCs

In this work, we propose the use of the *Distributed Object Model* (DOM) as the modelling framework for SoC applications. Under this philosophy, every component in a SoC is view as an object and communication is accomplished by means of *Remote Method Invocations* (RMI). The object providing the method or service to be invoked is called *server* and the one requesting such service is called *client*. Servers are passive whilst clients are active entities.

The concept of object is a powerful abstraction that effectively unifies the communication interfaces for both HW and SW elements in a SoC. Contrary to what is believed, objects do not impose any restriction on the way communication and synchronization can be modelled; message-passing, shared-memory, rendezvous and even threads and locks are artifacts that can be easily implemented in an object-oriented fashion [14]. To sum up, objects do not represent a design limitation at all.

3.1. Proxies and Skeletons: Efficient Decoupling of Behaviour and Communication

As mentioned before, client objects communicate with server objects by means of method invocations. A *method* is the access point to the object's functionality. A method consists of an *operation name* and, optionally, some *parameters* and/or a *return value*. The *remote interface* specifies the methods that are externally accessible for an object.

The distributed nature of the model considered in this work assumes that method invocation does not take place *locally* (which means point-to-point connections in a SoC) but through the use of some short of communication infrastructure. This assumption fits properly for SoCs platforms where communication between processing elements uses buses or Networks-on-Chip.

RMI relies on two simple constructions to make the object implementation independent of the communication infrastructure to be used: *proxies* and *skeletons*.

A proxy, which is placed between the client and the transport layer, implements exactly the same interface as the server does, providing the client with the illusion it is interacting with the actual server. The client invokes the proxy (figure 1, step 1) which builds a request message and sends it accordingly to the physical layer protocol (figure 1, step 2). Then, the skeleton

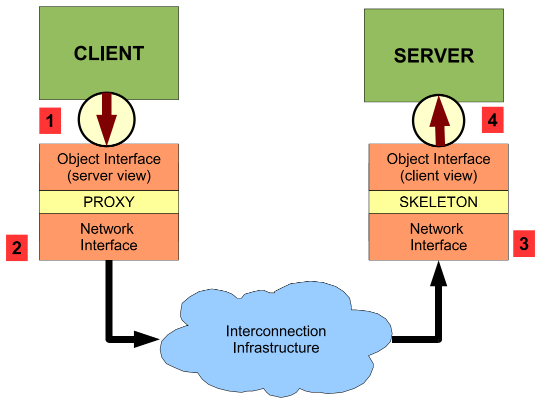
CLIENT
SERVER
1
4
Object Interface (server view)
PROXY
Network Interface
2
Object Interface (client view)
SKELETON
Network Interface
3
Interconnection Infrastructure

Figure 1: Four steps to remotely invoke a method in DOM.

receives the request message, interprets it (figure 1, step 3) and, finally, performs the real method invocation on the server (figure 1, step 4). Optionally, after the method of execution, a response message to communicate results or error conditions back to the client may take place.

In RMI, method invocations are translated into messages that must be delivered to the destination objects. To assure compatibility, data items must be packed following a predefined coding rules; this process is called *marshalling*. On the contrary, the *unmarshalling* process comprises the opposite steps to convert a planar data representation into one that can be understood by the target of the message.

To deliver a message to its destinations correctly, each object in the system has its own, unique object identification (OID) which is indispensable to address messages to servers.

3.2. Benefits of DOM to SoC Design

The application of the ideas, techniques, principles and architecture behind the DOM has been especially fruitful in the domain of networked computer systems. As in SoC design, heterogeneity and interoperability problems have been recurrent in networked systems for decades. However, the appearance of *middlewares* changed the way distributed applications were developed.

A middleware is an abstraction layer that hides the implementation details of the underlying platform so that communication can be managed in a *transparent way.* Transparency in SoCs would mean more freedom to the developers who could concentrate their effort on the essential facets of the solution so that better results could be obtained.

Table 1: Main types of transparency and their application to SoC design.

Type	Description	SoC application
• Access • Location	The way remote and local resources are accessed is identical It is not necessary to know where the resource resides	HW/SW interfacing, system co-design, unified view of the SoC, model reutilization, IP integration and replacement, robustness against changes, remote access
• Replication	Programmers do not have knowledge of the multiple instances of a resource	Enhances the reliability and the productivity, mechanism to balance the traffic
• Failure	Users and applications can complete their duties in spite of the hardware, network or software errors that may occur	Component replacement, increases the dependability
• Migration	Users and applications are not affected although clients and resources may be reallocated	Implementation of quality of service mechanisms, reconfiguration is easier, bug fix

Table 1 summarizes the most important types of transparency that it would be desirable to incorporate into a SoC. Table 1 also relates each type of transparency to the application and potential benefits in SoC design. As it can be seen, the most important types are *access* transparency and *location* transparency, both making the most relevant contributions. Ideally, the SoC designers should not worry about questions such as 'where the functionality will be located or implemented (HW or SW)' nor 'what on-chip communication infrastructure is going to be used'. Therefore, *transparency* is a key factor in OOCE and an enabler for a truly and uniform HW-SW interface abstraction.

DOM is a simple, well structured model that allows the automatic generation of the application-dependent parts which has also contributed to a quicker adoption of the middlewares based on this model. *Automation* is a must in SoC design because of the narrower market windows. Productivity levels would increase not only because of generation of the communication stubs but from less time spent in system verification (generation techniques use a *correct by construction approach*).

4. OOCE: The DOM Realization for SoCs

Although many analogies can be found between SoCs and networking distributed systems, not all the concepts and definitions used in current software middlewares can directly be applied to SoC design: *a think again* process is needed in which some of basic middleware ideas have to be revisited to make them valid in the new context (e.g. not all the implementation mechanisms used in SW are valid in HW since the design requirements, such as power or area, are stricter). OOCE is the result of such process, a system-level middleware for SoCs based on DOM and RMI to seamlessly integrate the HW and the SW SoC components.

The DOM defined by OOCE includes:

a) A recommendation of implementation of objects as hardware modules. This is necessary since there is not a clear correlation between the concept of object and a hardware implementation.
b) A specification of how invocations between objects within a SoC are mapped to read and write transactions over the interconnection infrastructure.

OOCE presents a hybrid communication infrastructure specially tailored for SoCs where many of the entities that conform the platform are implemented in HW so that maximum efficiency and low 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 high-level specification of the system.

4.1. 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 in order to help the generators of the OOCE communication adapters. This includes:

a) A standardized and simple interface to model point-to-point connections with the IP that implements the object behaviour.

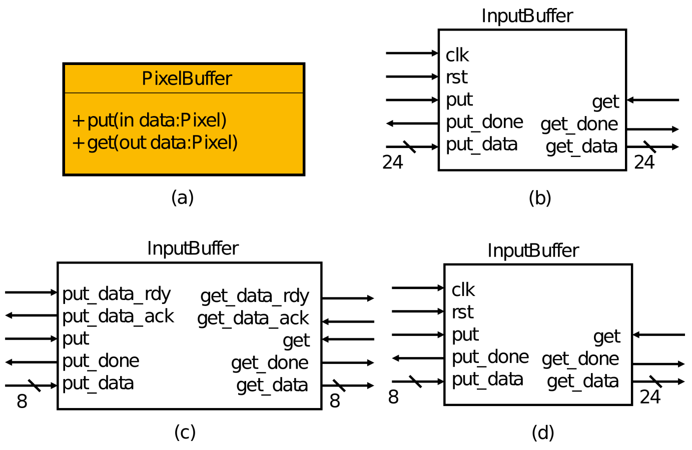

Figure 2: Hardware objects: (a) definition, (b,d) synchronous point-to-point protocol, (c) asynchronous protocol with the corresponding control signals for feeding data.

b) A *local* method invocation protocol. Basically, the interface signal activation sequence and temporization used to initiate an operation in a component and data transfer.

One of the main features of the hardware object model is the flexibility to define how values are fed to/retrieved from the IP and to define 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.

To illustrate the concept of hardware object, consider the example of Figure 2. A special memory buffer capable of storing pixels of an image is modelled in (a). Three implementations of the equivalent hardware core are depicted. The modules (b) and (d) implement a synchronous invocation scheme whereas in (c) an asynchronous one was chosen. A 24-bit port means the three pixel components are expected in one cycle whilst an 8-bit port will need three cycle or protocol steps to complete a pixel transaction.

4.2. OOCE Remote Method Invocation

OOCE defines a method invocation as a structure (see Figure 3) that contains the following items:

- A header with all the information needed for message addressing and delivering. The *Target OID* serves for component addressing and the *Operation Code* selects the method to be triggered in the server. A *Source OID* is needed only when the execution of the method produces

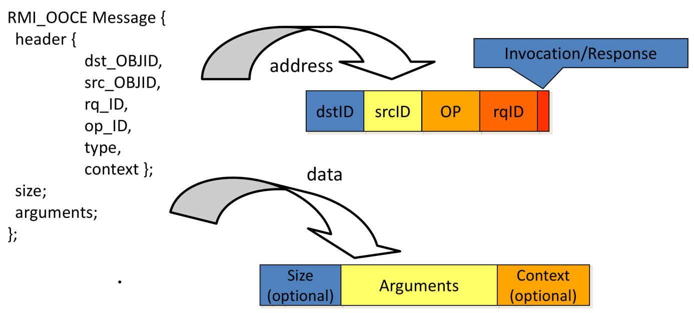

Figure 3: Mapping of an OOCE RMI message to an address-data pair.

a result or it may communicate an error condition. A *Request ID* is embedded for response reordering at the client side if out-of-order processing is allowed in the server.

- A body, which codifies the method *arguments* and the *context* according the OOCE data encoding rules. These rules are quite simple and compatible with ICE[1] (a CORBA-like middleware) protocol. Such compatibility enables easy and standard *off-chip* communication and system integration.

In a SoC, an invocation maps to a bus transaction targeting the object implementing the required functionality. The header fields are combined to form an address and the rest of the fields and body map to a bit-stream packed in words of the size of the data line width. Then a sequence of *write* operations on the destination delivers the message.

SW objects have their own address space in order to keep the communication model homogeneous and simple. OOCE infrastructure is responsible to make the SW invocation semantics compatible with the transaction base nature of the interconnection infrastructure (see 6.2). The format of the messages remains unchanged, what ever the nature of the communicating objects which 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.

The OOCE RMI protocol defines the number and type of messages that the client object must exchange with the server object to complete a method

[1]http://www.zeroc.com/

invocation in the latter. 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 client to 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.

In OOCE, there are no restrictions about the number of parameters a method may have or the number of output arguments. The supported data types in OOCE are: *basic* types (e.g. Boolean, byte, char, integer, long, float, etc.), *user-defined* types (structures with a combination of basic types), *sequences* (i.e. a string is a sequence of characters) and *buffers*. A buffer is a structure composed by a reference to a region in memory space, where data can be read or written, and a length.

OOCE can manage the invocation process both asynchronously or synchronously. The latter is indicated for low-latency one-way or two-way invocations whereas the former is intended only for two-way high latency methods. The asynchronous invocation semantics support provided by OOCE is an opportunity to easily develop high-performance computing techniques such as parallel method invocations in a pool of servers.

5. System Specification and Platform generation in OOCE

To offer a complete support to SoC design based on OOCE, a design workflow has been proposed. The starting point is an object model of the application to be implemented.

We use the *Unified Modelling Language* (UML) to capture the static view of the application. The designer annotates the UML entities with 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) whether the object's functionality can move from the SW to the HW domain or vice versa; (3) a specific communication scheme (synchronous or asynchronous); (4) whether the object belongs to a replica group or not (OOCE feature to enhance system reliability); or (5) whether the object is a static object or dynamic object (OOCE reconfiguration service). We wrote a *code analysis* tool that, from a C++ *application*, does get its object model. The designer can also write such an object model if the reference software application is not available.

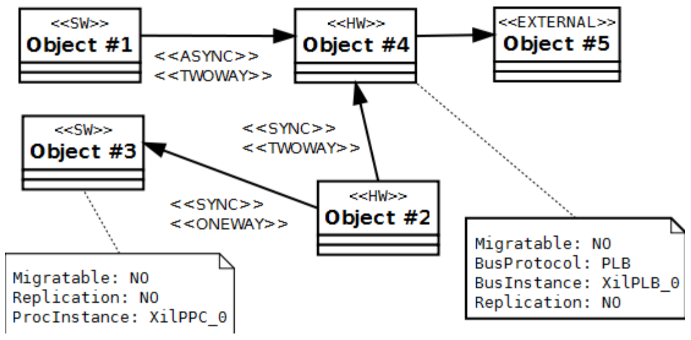

Figure 4: Example of a UML OOCE annotated diagram.

The mapping of the objects to platform resources is implicit in the annotating process since stereotype attributes are used to link application objects to entities in the platform. For the experimental prototypes, we have modelled the Xilinx ML505 board in a separate UML component diagram using the MARTE standard [15]. The *XilML505* profile extends some of the basic MARTE stereotypes in order to facilitate the platform generation as a *Xilinx XPS project*.

As an example, Figure 4 shows an application with five objects where Object #3 is tagged to run on processor instance XilPPC_0 by means of the *ProcInstance* stereotype attribute. For objects tagged with the ≪HW≫ stereotype the bus instance where it will be connected is indicated. In this example, for the sake of simplicity, method signatures for every object have been removed in figure 4 and object relations have been tagged with the *oneway* or *twoway* stereotypes. In an actual specification diagram, such information is derived from the method signatures that each object implements (with or without return/output parameters, respectively).

The textual representation of the object diagram feeds: (1) a HW interface compiler which generates the OOCE HW adapters; (2) a SW interface compiler which generates the OOCE software adapters; (3) an OOCE platform generator which selects, from a component template library, the communication engine components required by the application; and finally, (4) a program that combines all the above mentioned elements and generates the XPS project.

At this point, the designer obtains a complete prototyping platform which is ready to be synthesized using Xilinx EDK standards tools. Figure 6 sketches the derived HW and SW infrastructure from the OOCE UML an-

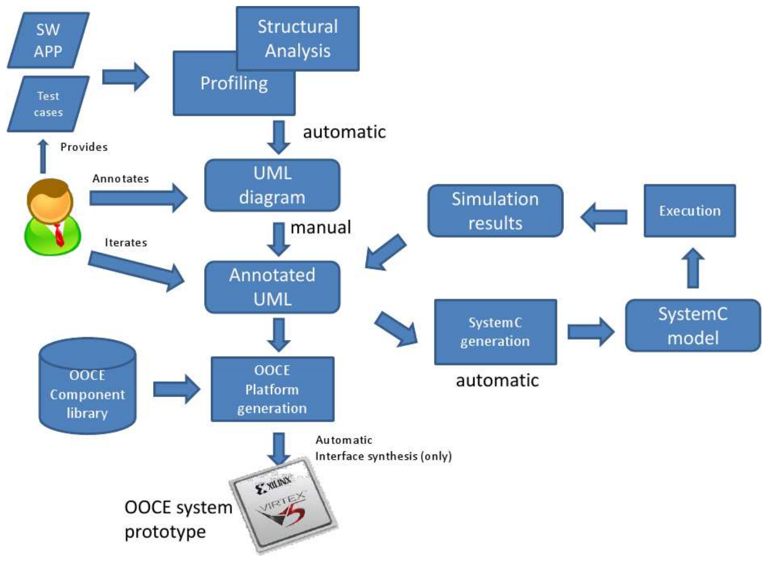

Figure 5: Example of a UML OOCE annotated diagram.

notated diagram of Figure 4. The designer should only: (a) connect the HW cores implementing the OOCE hardware objects #3 and #2, following the standard module interface and activation protocol. These components could also be retrieved from an existing OOCE compliant IP library, or a legacy one may be easily adapted; and (b) re-write the behavior of the client application using the generated OOCE SW drivers. This process is simple and does not require complex changes in the original code.

To help the designer in the exploration of the design space, a SystemC model of the platform is generated for simulation and verification purposes. The goal is to provide the designer with a quick tool to detect potential communication bottlenecks, and estimate the required bandwidth. The profiling information obtained from the running of the model is used to help in the deployment process and the selection of a HW or SW implementation for a given object. This configures an iterative design framework (Figure 5) that progressively reaches a heterogeneous implementation. This is possible thanks to the access and location transparency principles supported by OOCE. In brief, the steps followed in the whole process can be summarized as follows:

- Code Analysis and profiling (automatic). The goals in this stage are: (a) obtain the UML diagram of the static view of the application (objects and relations); and (b) characterization of execution times for the application that will be incorporate to the SystemC model later.

- Analysis and annotations (manual). Performed by the designer. It is possible here to adjust some of the parameters obtained from profiling and assign objects to functional units (hardware-software) and configure the communication methods between them.
- Generation of the SystemC model (automatic).
- Generation of the prototyping platform (automatic). Only interface synthesis.

The current version of the SystemC models is intended to provide fast results rather than accurate ones. Nevertheless, the simulation numbers obtained are quite close to the actual ones. This is mainly due to the simplicity, determinism and regularity of, for example, the hardware state machines that implement the proxies, skeletons and the point-to-point invocation. Also, the SystemC version of the proxies and skeletons is almost the same as that obtained using the platform interface compilers. All this together makes it easy to know the number of cycles the processing/generation of an OOCE RMI message takes in advance. To simulate the bus infrastructure, the GreenBus[2] SystemC framework has been used whereas the OOCE HW/SW interfacing infrastructure has been modelled partially in SystemC, skipping the OS dependent layers.

6. The Object-Oriented Communication Engine in Detail

Besides the introduction of the proxies and the skeletons, the OOCE architecture needs additional elements and a hybrid infrastructure to support the three communication scenarios. All of them are depicted in Figure 6.

It is worth emphasising the fact that automatic generation is available both for hardware and software versions of proxies and skeletons. The rest of the OOCE components are platform dependent and must be written *just once* for every new platform. For example, the *Kernel Local Object Adapter* (KLOA) which is OS dependent or the *Local Network Interface* (LNI) that depends on the connection with the processor.

[2]http://www.greenbus.com

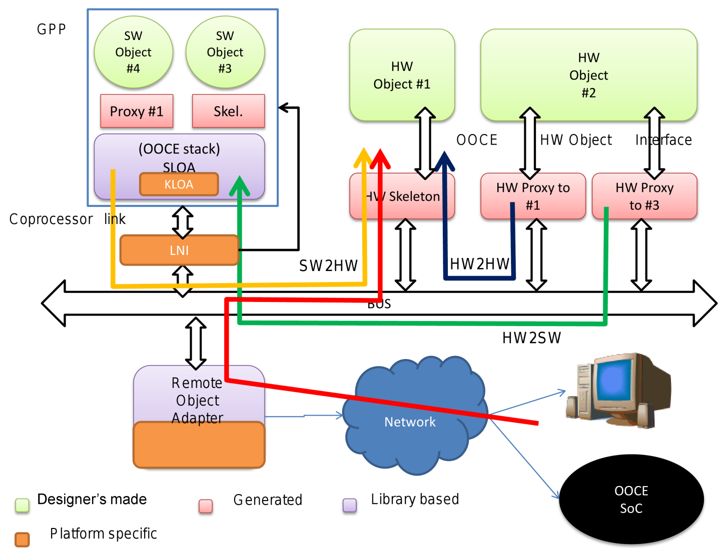

Figure 6: Main components of the OOCE architecture.

Throughout this section these components and their role in the different communication scenarios will be the object of analysis.

6.1. HW to HW invocation

HW to HW invocations can take place between HW objects in OOCE. No action from the OS is required, freeing the processor from control tasks. The HW objects are isolated from the communication infrastructure by means of the proxy and skeleton wrappers. Client and server HW objects are provided with the illusion they are still point-to-point connected whereas local invocations are forwarded through the bus.

The general architecture of a proxy interface or skeleton (Figure 7 shows the block diagram of a HW skeleton) comprises the following layers:

- The *Adaptation Unit*. It is the first level of isolation. To make the proxy/skeleton architecture more portable and modularized, bus specific control signals are translated to a bus independent read/write protocol.

- The *Control Logic* (CL) implements the Finite State Machine (FSM) in charge of the local and remote protocol adaptation.

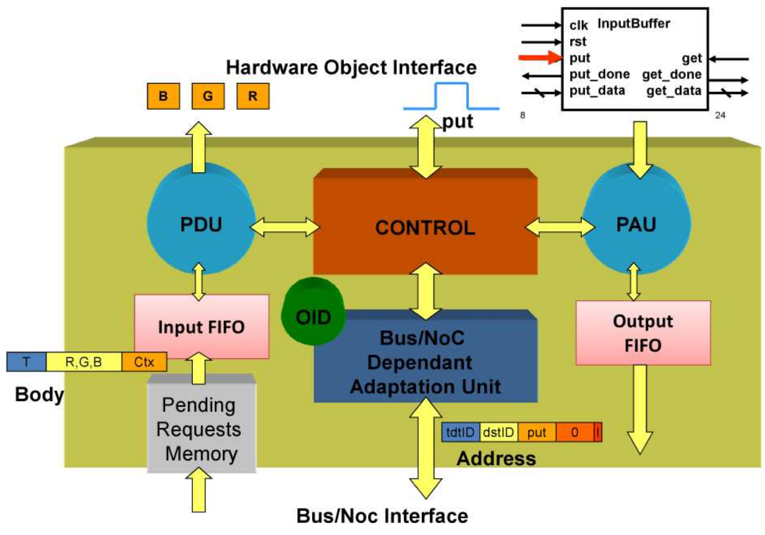

Figure 7: The skeleton template for HW method invocation. Both synchronous and asynchronous communications are considered.

- The marshalling and the unmarshalling processes are carried out by the *Port Acquisition Unit* (PAU) and the *Port Delivery Unit* (PDU). This modules are generated automatically from the description of the method they serve.

- Two FIFOs are required to temporarily store input and output OOCE RMI messages. All elements in this layer are customized according to the signature of the methods offered, so the resulting implementation is optimal. For example, a memory for pending requests would not be included in the case of synchronous communication.

This modularity enables better opportunities for reutilization and portability of the solution. For example, a new proxy/skeleton for a different interconnection technology is easily obtained exchanging the ”Adaptation Unit”. Or the same proxy/skeleton template can serve a different method only exchanging the PDA and PAU modules, that are generated by the interface compilers.

Synchronous or asynchronous communication and direct or indirect communication (see 6.3 for some considerations on indirect communication) is supported in OOCE. As the result of the combination of these parameters, several templates have been defined. These templates are the inputs to the

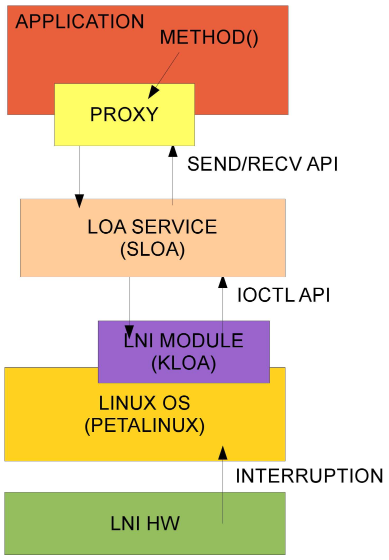

Figure 8: OOCE software stack

OOCE HW generators.

6.2. HW/SW interfacing

The OOCE RMI protocol ensures that any method invocation involving HW and SW objects uses exactly the same messages as those generated in a HW to HW method call. Due to this the HW templates for proxies and skeletons need no modifications. The OOCE infrastructure for transparently handling HW/SW interfacing is minimal.

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 holds a *Translation Address Table* (TAT) to route the relevant bus traffic to the processor. If a SW object wants to be reachable from outside the processor, it must register its object identifier in the TAT.

In order to abstract the access to the LNI a three layer software architecture has been defined. Figure 8 represents such architecture. In our prototypes only Linux based OS have been considered.

6.2.1. Kernel Local Object Adapter

The KLOA is the first layer, implemented as a kernel module of the OS. It can be considered LNI driver in the system. By means of the use of *ioctl* primitives applications in user space can access the LNI to send or receive messages and manage the TAT. Nonetheless, it is not accessed directly by the user applications but through the next layer in the software stack.

KLOA main functionality consists in attending HW interruptions from the LNI module. An interruption is fired when the LNI signals a message is ready in the Rx FIFO or when a message pushed into the Tx FIFO has been sent. Incoming messages are buffered until the *Service Local Object Adapter*, the next level in the hierarchy, is ready to process them.

6.2.2. Service Local Object Adapter

The SLOA is implemented in user space and makes use of the KLOA interface to manage the message traffic between applications and HW cores. In the case of SW-to-HW messages, it implements the necessary contention mechanisms to assure a fair and safe use of the LNI (which is unique in the system) from the software objects running in the processor. Each HW-to-SW message delivery is handled by a different thread to enhance reliability, avoiding undesirable side effects due to misbehavior of the destination objects. In few words, the SLOA multiplexes and demultiplexes the traffic between the software objects and the hardware entities.

The proxies and skeletons in the user application use a message-passing interface to communicate with the SLOA. The clients must be provided with the port and protocol the SLOA is running at (as command line arguments or using a system configuration file).

6.2.3. Proxies and Skeletons

Embedded software programmers only have to know about software proxies and skeletons in order to interact with the middleware. The rest of the infrastructure is hidden below these automatically generated software routines. A SW proxy links the object facade to the HW device (figure 9) making applications more robust to unforeseen changes in the platform. A SW skeleton is a callback function activated by the SLOA once it has been selected as the target of an incoming request. To do this, the SLOA holds a table that relates the object identifiers with the reference (a pointer) to the skeleton that will attend the request.

```
void DES_crypt(tPrxy *prxy,long key,long data,
               long *encrypted) {
  tOOCE_msg msg;
  int sock, recv_code;

  msg.src = prxy->m_objid;  /* header, source ID */
  msg.dst = prxy->objid; /* destination ID */
  msg.rid = prxy->rid++; /* request ID  */
  msg.op = DES_CRYPT_MID; /* Operation code */
  msg.type = OOCE_MSG_TWOWAY; /* there is a response */
  msg.size = 4; /* total size of the msg in words /
  *(long *)(msg.data+0) = key;  /*marshalling*/
  *(long *)(msg.data+8) = data;

  /* … socket creation and connection to the SLOA.
    Send the message. */
  send(sock, &msg, OOCE_HEADER + msg.size, 0);

  /* wait for the response */
  recv_code = recv(sock, &msg, sizeof(tOOCE),0);

  /* unmarshalling */
  (*encrypted) = *(long *)(msg.data+0);

  return;
}
```

Figure 9: Example of a SW proxy for an encryption method (DES object).

6.3. Advanced middleware features

Now, we give a brief introduction to some OOCE advanced services and applications which have been built upon the basic communication facilities. A HW implementation of the mechanisms and components hereafter described is provided so as to obtain the best performance.

a) *System-to-System integration*. OOCE provides transparent off-chip communication with external components implementing the ICE (an object-oriented commercial middleware widely used in the industry) protocol. This allows OOCE SoCs to interact either with ICE servers running in a PC or other OOCE SoCs in the same object-oriented approach. The *Remote Object Adapter* (ROA) is the OOCE component responsible for offering such functionality. Since in-chip packet format does not change, in-chip middleware infrastructure does not have to be modified in order to fit in the new scenario.

b) *Location service* (LS). OOCE allows the use of a logical reference to invoke an object instead of its physical base address; indirect communication (IC). IC is the basis for more complex services and applications (see later in this section). The LS is used to translate the logical references into in-chip access addresses.

c) *Group Communication* (GC). OOCE can, with guarantees, deliver one invocation message to several destination objects with only one bus transaction. Group invocations are regular OOCE RMI using a *group identifier* as the destination. The implementation of a subsystem to communicate exceptions (errors at the application level) or events and a service discovery protocol (applied to reconfigurable objects) are the two GC main applications.
d) *Synchronization component library* (SCL). We have developed a HW version of mutexes, semaphores and mailboxes to easily adapt pre-existing concurrent applications.
e) *Reconfiguration Service.* The use of IC in adaptive, dynamic applications (using dynamic reconfigurable logic) can provide important advantages. Details of a reconfiguration service based on this paradigm can be seen in [16].
f) *Run-time failure management.* The LS may hold several physical references for a set of objects that implement the same functionality. The LS provides a new valid reference from that set if an error condition is detected. The replacement object can be indistinctly implemented in SW or HW.
g) *Migration.* It may be necessary that functionality "crosses" the SW and HW boundaries (even the chip boundaries) under certain conditions. The system objects that are susceptible to be migrated must implement a persistence interface to save their state and recover it later. IC along with the HW/SW transparency offered by the LNI-LOA components enables this scenario.
h) *Quality of Service and load balancing.* The LS can store statistics on how often an object is accessed. If this object is replicated in the system, the LS can decide if new accesses are forwarded to the less used replicas.

7. Experimental Validation

To provide evidence of the concept of our proposal, we have implemented the OOCE architecture, methods and tools using the Xilinx ML505 prototyping platform. Two versions have been considered, one for each type of processor supported by this board: *Microblaze* soft-processor running *Petalinux v0.3-rc1* and *PowerPC 405* running a *Linux-2.6 kernel.*

First, the Microblaze version was developed and the adaptation to the PowerPC involved only two designers during two weeks. Only the platform dependent infrastructure needed changes, making the porting procedure easier. The work performed includes: (a) a new version of the LNI, (b) an extended version of the interface compilers in order to support PLB buses (in addition to the already existing OPB generators) and (c) minimal changes in the software stack due to slight differences between the two Linux platforms.

Table 2: Synthesis of results (OPB only)

Metric	DES	AES	CORDIC	SOBEL	PREWITT
IPIF FFs	1432	1245	404	734	888
OOCE FFs	844	896	230	609	613
FFs %Diff.	**-41%**	−28%	−43%	−17%	−31%
IPIF LUTs	2312	2081	630	1093	1180
OOCE LUTs	1642	1706	498	1016	1026
LUTs %Diff.	−29%	−18%	−21%	−7%	−13%
IPIF Dev. time (hours)	24	22	27	38	35
OOCE Dev. time (hours)	19	18.5	19.5	23	24.5
Dev. Time Diff %	−21%	−18%	−33%	−39%	−30%

From Table 2, it can be observed that the evaluation of the extra HW resources needed by the OOCE infrastructure is quite satisfactory. We have compared the total amount of logic of five legacy cores and the development time to make them usable as OPB peripherals. OOCE components show a decrease in the resource demand against their equivalent solution using Xilinx IPIF. The numbers in Table 2 for OOCE results do not include the logic consumed by the LNI but, since it represents about 1% of the available on board resources and it can be shared among all the cores in the system, the actual variation is marginal. The number of development hours we are referring in Table 2 comprises the time spent in: (a) writing the glue logic to make the core compatible with the bus wrapper, (b) simulations for verification purposes, (c) writing the drivers and application tests and, (d) final tests on the actual platform. The savings in development time are mainly due to the fact that OOCE tool chain already provided the designer with the drivers, reducing the time spent in this step and the related verification and test cases to the minimum.

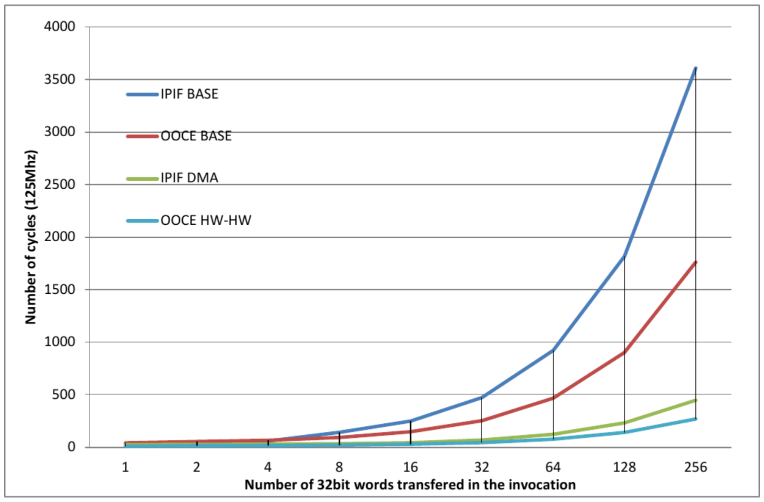

Figure 10: On-chip transaction latency using Xilinx IPIF and OOCE.

To finish the revision of the experimental results regarding resource usage and development times, it is worth mentioning that in both cases (IPIF and OOCE) it was reached the target frequency expected for the system (125 Mhz). In regard to the overhead in area caused by the OOCE components needed to provide in-chip and off-chip communication it is worth mentioning that the LNI requires only 46 flip-flops, 148 LUTs and 83 Slices in a LX110T Virtex5 chip (less than 1% of the available resources) and can work at a frequency near to 340Mhz. The Remote Object Adapter represents (recall it is only present if the off-chip invocation feature is requested) 10% of the resources in board (1116 flip-flops, 3116 LUTs and 1692 Slices) and runs at 100 Mhz.

Since communication efficiency is crucial in typical mixed hardware-software computation scenarios, we have measured the overhead introduced by the proposed infrastructure and the software stack. Two scenarios have been considered.

The first one is intended to compare the conventional memory mapped communication for IPIF and OOCE. To this end, data is written in series of 1 to 256 words using a software routine for IPIF or a dummy software proxy for OOCE. After completion, the processed result is read from the core. The test platform comprises the Microblaze soft processor, a timer to accurately measure the cycles spent in the process, a simple core that performs the sum of the words written to it and the PLB bus. In figure 10 functions labeled

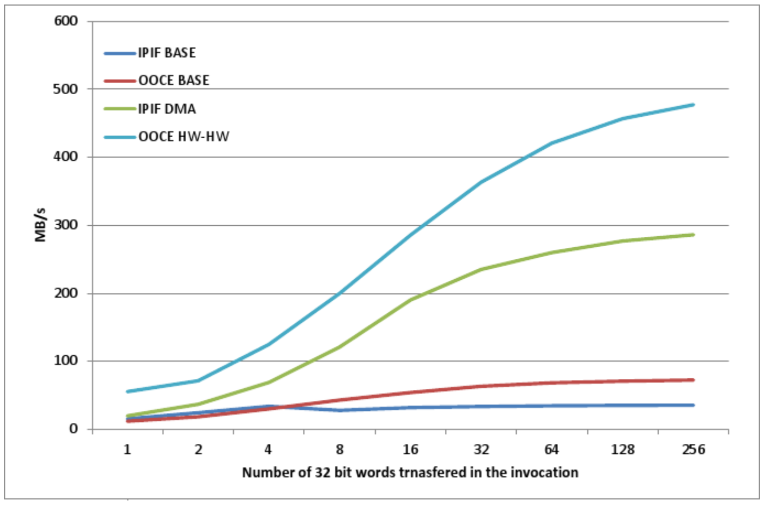

Figure 11: On-chip transaction throughput using Xilinx IPIF and OOCE.

IPIF BASE and *OOCE BASE* represents the behaviour of the two alternatives for this scenario. For transactions above 16 words OOCE behaves better reaching a gain between 40%-50% because the middleware handles data movement between the LNI and the core by means of bursts instead of the one word at a time scheme imposed by IPIF. Below 16 words, the penalty introduced by the middleware infrastructure makes it the communication times comparable. Time is expressed in term of number of cycles which actually means a precision of nanoseconds.

The second test case makes it use of the DMA facilities for IPIF and the HW to HW semantics for OOCE. As in the first scenario, direct transfers between two hardware cores have been programmed and only the time spent in data transmission have been measured. Configuration routines and preparation of data were not considered. The results depicted in 10 confirm the good behavior of OOCE in this scenario as well. IPIF DMA wrappers generated with the Xilinx tools cannot perform burst transfers of 32 words or more which represents a clear limitation not present in OOCE wrappers.

Finally, figure 11 shows the maximum data transfer rate achievable for each of the four tests. When the number of data to be exchanged is high enough, OOCE mechanisms double the numbers obtained with IPIF.

8. 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 modelled as communicating objects.

The principal features of OOCE are: (1) most of its components are generated in an automatic way; (2) it is flexible since it is extremely easy to adapt it to new target technologies; (3) it provides the same programming interface for HW and SW elements (which boost the productivity of the embedded software developers); (4) it adds the necessary semantics to directly translate invocations to an implementation level using elemental communication services; and (5) it supports advanced services to ease the management of complex tasks such as synchronization, migration, replication, etc.

[1] A. Jerraya, HW/SW Implementation from Abstract Architecture Models, in: Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07, 2007, pp. 1 –2. doi:10.1109/DATE.2007.364507.

[2] M. Vuletid, L. Pozzi, P. Ienne, Seamless hardware-software integration in reconfigurable computing systems, Design Test of Computers, IEEE 22 (2) (2005) 102 – 113. doi:10.1109/MDT.2005.44.

[3] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, Y.-P. Joo, Peace: A hardware-software codesign environment for multimedia embedded systems, ACM Trans. Des. Autom. Electron. Syst. 12 (2008) 24:1–24:25. doi:http://doi.acm.org/10.1145/1255456.1255461.
URL `http://doi.acm.org/10.1145/1255456.1255461`

[4] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins, Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip, in: Design, Automation and Test in Europe Conference and Exhibition, 2003, 2003, pp. 986 – 991. doi:10.1109/DATE.2003.1253733.

[5] J.-C. Chiu, T.-L. Yeh, Ires: An integrated software and hardware interface framework for reconfigurable embedded system, Computers Digital Techniques, IET 4 (1) (2010) 27 –37. doi:10.1049/iet-cdt.2009.0010.

[6] H. K.-H. So, A. Tkachenko, R. Brodersen, A unified hardware/software runtime environment for FPGA-based reconfigurable

computers using BORPH, in: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis, CODES+ISSS '06, ACM, New York, NY, USA, 2006, pp. 259–264. doi:http://doi.acm.org/10.1145/1176254.1176316.
URL `http://doi.acm.org/10.1145/1176254.1176316`

[7] A. Donlin, P. Lysaght, B. Blodget, G. Troeger, A virtual file system for dynamically reconfigurable fpgas, in: Field Programmable Logic and Application, Lecture Notes in Computer Science.

[8] H. Lange, A. Koch, Architectures and execution models for hardware/software compilation and their system-level realization, Computers, IEEE Transactions on 59 (10) (2010) 1363 –1377. doi:10.1109/TC.2009.180.

[9] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagne, G. Nicolescu, Parallel programming models for a multiprocessor soc platform applied to networking and multimedia, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 14 (7) (2006) 667 –680. doi:10.1109/TVLSI.2006.878259.

[10] W. Klingauf, R. Günzel, C. Schröder, Embedded software development on top of transaction-level models, in: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis, CODES+ISSS '07, ACM, New York, NY, USA, 2007, pp. 27–32. doi:http://doi.acm.org/10.1145/1289816.1289827.
URL `http://doi.acm.org/10.1145/1289816.1289827`

[11] G. Gailliard, H. Balp, M. Sarlotte, F. Verdier, Mapping semantics of corba idl and giop to open core protocol for portability and interoperability of sdr waveform components, in: Proceedings of the conference on Design, automation and test in Europe, DATE '08, ACM, New York, NY, USA, 2008, pp. 330–335. doi:http://doi.acm.org/10.1145/1403375.1403455.
URL `http://doi.acm.org/10.1145/1403375.1403455`

[12] K. Hao, F. Xie, Componentizing hardware/software interface design, in: Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09., 2009, pp. 232 –237.

[13] G. Gailliard, H. Balp, C. Jouvray, F. Verdier, ”towards a common hw/sw interface-centric and component-oriented specification and design methodology.”, in: Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on, 2008, pp. 31–36.

[14] B. P. Douglass, Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[15] O. M. Group, Uml profile for modeling and analysis of real-time and embedded systems, Standard specification, OMG (Nov 2009).

[16] J. Dondo, F. Rincon, J. Barba, F. Moya, F. Villanueva, D. Villa, J. Lopez, Dynamic reconfiguration management based on a distributed object model, in: Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on, 2007, pp. 684 –687. doi:10.1109/FPL.2007.4380745.