System-Level Middleware for Embedded Hardware and Software Communication

Fernando Rincón, Jesús Barba, Francisco Moya, Félix J. Villanueva,
David Villa, Julio Dondo, Juan Carlos López
School of Computer Engineering
University of Castilla-La Mancha, Ciudad Real, Spain
{fernando.rincon, jesus.barba, francisco.moya, felix.villanueva,
david.villa, julio.dondo, juancarlos.lopez}@uclm.es

Abstract *— Heterogeneous system architectures are currently the main platform on which an ever increasing number of innovative applications (i.e. smart home or ambient intelligence applications) rely. When designing these complex systems, one of the most time-consuming tasks is the definition of the communication interfaces between the different components through a number of scattered heterogeneous processing nodes. That is not only a complex task, but also very specific for a particular implementation, which may limit the flexibility of the system, and makes the solutions difficult to reuse.*

In this paper, we describe how to provide a unified abstraction for both hardware and software components that have to cooperate with each other, independently of their implementation and their location. Based on this abstraction, we define a low-overhead system-wide communication architecture that offers total communication transparency between any kind of components. Since the architecture is highly compatible with standard object-oriented distributed software systems, it also enables seamless interaction with any other kind of external network.

1 Introduction

Latest consumer applications (e.g. multimedia processing or 3D games) demand complex designs to meet their real-time requirements while respecting other design constraints, such as low-power or short time-to-market. In this context, Systems-on-Chips (MPSoCs) have been proposed as a promising solution. Nevertheless, one major challenge in such systems is the integration in the platform of the multiple Application Programming Interfaces (API) that each component (e.g. memory, buses, cores, etc.) is designed for. Moreover, another important problem in SoC design is the knowledge of the position of each component in the final system to be able to efficiently communicate with it (e.g. local, remote), which makes the correct design of a SoC even more complex.. Thus, new methods that allow designers to get unified inter-communication methods on SoCs architectures in the system integration flow are in great need.

Some concepts taken from distributed object platforms such as CORBA or Java RMI have already been applied to SoC design in order to get a unified view of HW and SW modules. In this paper we present an approach which inherits most of these previous

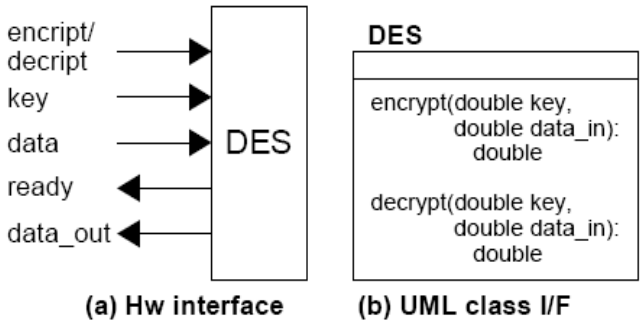

Figure 1: DES HW interface

achievements enriched with a strong focus on location transparency and network transparency. The resulting architecture provides a unified view of the whole system and also enables the designer to seamless develop multi-SoC systems with different network technologies.

This paper is organized as follows. In Section 2 we present a motivational example that will serve to guide us through our proposed approach to homogeneous hardware and software modeling. In section 3 we present the overall hardware SoC architecture proposed for an efficient unified components interconnection. In section 4 we show some experimental results. In section 5 we overview some related work. Finally, section 6, summarizes the contributions of the paper and presents possible future research directions.

2 Motivational Example

Let's consider the design of a System on Chip for applications with some cryptographic requirements. For such purpose, the system will include a third party DES IP core, that is able to provide encryption an decryption of certain blocks of data. We can consider three different usage scenarios of the DES core, which are: 1) locally to the SoC, from another HW core, 2) also locally, but from a software client, and 3) from an external (to the SoC) entity (a pervasive computing application that needs some cyphering, for example). In the following paragraphs we will first describe each of the situations, next we will analyze what is the current approach to the design problem, and finally we will provide an alternative solution.

For the example we will use one of the DES cores provided by Opencores [1], and that will also be used to illustrate the experimental results in section 4.

2.1 HW-HW Communication

The first scenario will be the use of the DES IP core from another hardware component. The DES core obtained from Opencores has a very simple interface (figure 1a) with an encrypt and decrypt signal, a bus for providing the key, and the input and output data buffers.

Let's suppose that the SoC is using an OCP bus for the interconnection of the cores and processors. The first task would be to adapt it to that concrete bus. For such purpose we could use the *CoreCreator* tools from the OCP suite, and automatically generate the OCP wrapper out of a simple description of the DES interface. However, there is still some work to do, since we need to serialize the reception of the key and data input block, and the transmission of the data output block, since they don't fit in the bus word size[1].

On the other side, for the core using the DES, we should follow a similar but inverse procedure. Once known the bus interface, and the transaction-level protocol for for providing the data and obtaining the results, we could write the functionality of the client core, next include some logic for the serialization of the transmission, and wrap it automatically to the OCP bus. This is a very normal procedure for IP integration, where both components are attached to the bus through some bus adapters (wrappers).

One of the advantages of using bus standards such as OCP is that they ease reusability of previous designs. However, interoperability at the level of operations is not guaranteed by the standard. The definition of a certain order of the key and data arguments is hard-coded inside the wrapper. Also the way the arguments are divided into bus-size words is completely implementation dependent. That is the reason why cores with the same interface may not be interoperable.

All those problems are due to the loose coupling between functionality and communication. A well-known solution to this, is the orthogonalization of both aspects, thanks to the definition of a certain communication semantic at the transaction level communication. This semantic will abtract any operation in the system as a function call, and will in fact consist in the definition of:

- how the address for the transactions in the bus is build upon the target and the operation requested.
- how the arguments are ordered, so all requests for the same operation are always performed the same way, with independence of the source..
- how data types are serialized for their transmission through the bus.

However, our proposal differs from previous approaches, such as TLM, in the sense that the purpose is not only to abstract channels of communication and data types, but to fix the way communication flows. While such kind of abstractions are complex to synthesize since they depend on the expressiveness of the designer, a more restrictive semantic would define a straightforward path to implementation even for more abstract interfaces, as will be discussed later. Once the set of functions provided by a component are known, the interface is completely defined and the adapters on both sides can be fully automatically generated,

Let's return to the example, and let's reconsider it under the new assumptions. First we need to abstract the interface of the DES component to the level of functions (or class methods), which may be the one in figure 1b. Notice how only the operations, arguments and return values are included, and how they match with the DES entity. Next we need to

[1] We are supposing a 32 bits bus width

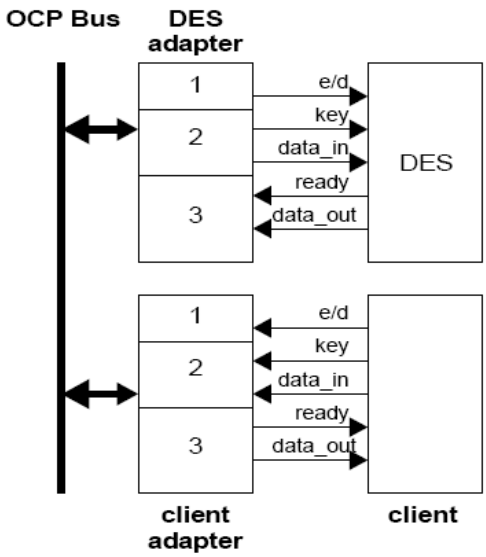

Figure 2: Client, DES communication

adapt the component to the OCP bus

Here the tasks are exactly the same than the ones previously described. The result for the DES core would be the one shown in figure 2. The adapter has 2 different interfaces, on one side it is connected to the OCP bus, on the other side it is linked to the component providing the functionality. Internally it includes 3 logic blocks:

1. the OCP address into decrypt/encrypt signals activation.
2. the OCP input data recovery state machine, which recovers they key and the data to cypher/decypher from the bus transaction, and routes it to the *key* and *data_in* ports of the component
3. the *data_out* serialization state machine, that injects the result as a return value of the OCP bus transaction.

The main difference between this approach and the initial one is that, this time, the rules for the sorting and serialization of the parameters and return values is predefined, and the logic in the blocks is deterministic. It is not difficult then to define certain templates and rules for the automatic generation of the adapter.

On the client side, the tasks and the interface of the adapter are exactly the opposite ones. On one side it interfaces the OCP bus with the standard signals, on the other side it provides the client component the DES interface. The logic of the adapter is structured in the same three blocks than the DES adapter, with the opposite functionality. This adapter will again be deterministic, once the interface of the DES component has been fixed, and hence also automatically generated.

Something that is worth noting is that the interface of the adapter with the client is exactly the same from the real DES component. Thus, it would be possible to connect the DES component directly to the client if we didn't have to share it with other IPs through the bus. So the client can be designed as if locally connected to the DES. In other words, neither the client nor the DES core need really to know that they interact to each other

```
class DES {
  public:
    double encrypt(double key, double data_in) {
      -- the object identity and operation identity are
      -- mapped onto an address
      putfsl(OBJ_ID << 16 + ENC_ID);
      -- the bus interface needs to know the number of
      -- arguments
      putfsl(NON_VOID | ENC_ARGS);
      -- arguments and return values are serialized as two
      -- 32-bit words.
      putfsl(key & 0xFFFFFFFF);
      putfsl(key >> 32);
      putfsl(data_in & 0xFFFFFFFF);
      putfsl(data_in >> 32);
      getfsl(data_out_low);
      getfsl(data_out_high);
      return data_out_high << 32 + data_out_low;
    }
    double encrypt(double key, double data_in) {
        ...
        ...
    }
```

Figure 3:C++ code for the sw client adapter

through certain adapters, since they really look and behave as the true core components.

2.2 HW-SW Communication

As a second scenario, we will consider how the DES could be used from a software client. To do so, in the classical approach we would need some kind of driver or API interface. These APIs are very dependent of the concrete core, and not easy to generate automatically. Even one minor change such as the modification of the address of the target component, or the order in which arguments are transmitted would require major modifications in the code.

The solution proposed is based on an homogeneous treatment of hardware and software resources. The main idea is to generate the same transactions in the bus than that generated by the hardware client. From the DES point of view there will be no way to distinguish if the source of the transaction is either a software or a hardware client.

From the implementation point of view that means that we also need an adapter for the software client. This adapter will again offer to the client the DES abstract interface, with the same operations, arguments and return values. So it will be completely transparent to the SW client if the DES API that it is using is implemented in hardware or software. The software version of the adapter will provide a similar functionality than the hardware counterpart (figure 3), and will be the responsible for generating the same bus traffic for the execution of the cypher/decypher operation.

Something to note is that using the DES core from a software client, in this approach, does not require a single modification of the core already used in the first scenario.

2.3 Remote Communication

The third scenario will correspond to the request of the cyphering from outside the SoC. In this case we will suppose that we have an ethernet interface with the outside.

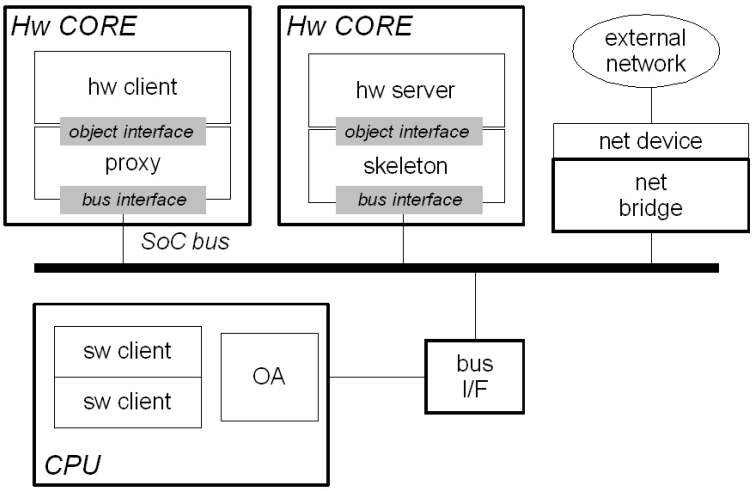

Figure 4: System-level middleware

This scenario is not very common, mainly due to the difficulty of multiplexing the ethernet between several components plus the microprocessors, which normally act as the masters of the device. Even, it is not clear how to translate network packets into the required bus transactions for the arguments and results of the operations. It would, however, be very useful to have remote communication to and from external clients, for debugging purposes, remote configuration, or even remote reconfiguration of the SoC.

In the proposed approach, remote communication does not imply nothing but a special bridge connected to the ethernet adapter. As it happens with SW to HW communication, interoperation is guaranteed by the use of the same bus transactions for the same operation requests. Thus, once a network packet coming from the outside reaches the bridge, it is injected in the SoC bus just if it was generated locally. The target core will recognize the address and perform the required operation putting back the results in the bus. These results are translated in the bridge back into network packages, and sent back to the remote client.

It is also possible to execute operations from remote servers from both hardware and software clients. They simply need the corresponding adapter (with the target server interface) that will translate operation requests into bus transactions. However, this transactions will not correspond to any address in the SoC address space, but will be mapped to the bridge. So the bridge will pack them into network packets, after a translation of the local SoC address to a network address (protocol and port), and will route them through the ethernet device.

3 The System-Level Middleware

Once the three different sceneries of communication have been described, and once a solution to the interoperation of elements has been proposed, we will describe the solution from a global point of view. Figure 4 shows the main elements that are part of

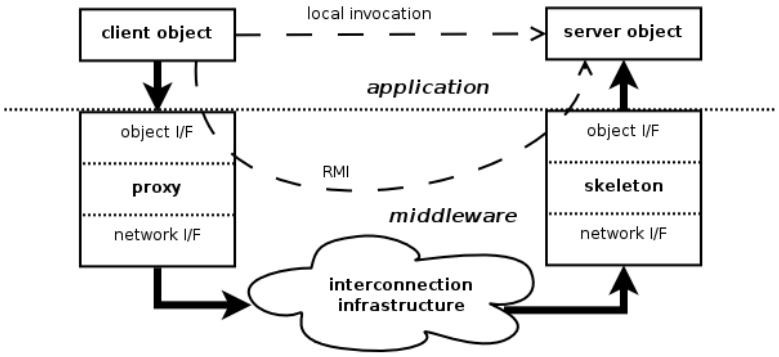

Figure 5: Actors in a Remote Method Invocation

the System Level Middleware presented in this work, and that are briefly described in the following subsections.

3.1 The SoC as a Distributed System

The ideas presented in the motivational example are inspired in the way software middlewares work for distributed object-oriented systems. A middleware is an abstraction layer whose main objective is to make homogeneous the communication between the components of a distributed system. Generally, a middleware bases its functionality on : (a) a client-server model of communication, (b) a common data type system and a set of data coding/encoding rules, and (c) a simple protocol defining the set of messages client and server exchange. The objective is to provide orthogonalization between behavior and communication.

Applications using the middleware are usually based on the object-oriented programming model. Objects also rely on a simple communication model: method invocation. This same mechanism is used for remote communication (Remote Method Invocation or RMI), where invocations are translated into synchronous messages passed though a certain communication infrastructure. The main advantage of RMI is that it provides a neat separation between functionality and communication. That makes Distributed Object Systems specially suited to deal with heterogeneity and scalability of applications.

Figure 5 shows the actors in a remote communication and the domain they belong to (application or middleware). Any method invocation must take place between a Proxy and a Skeleton. From the client point of view, the proxy is the requested object itself, since it provides exactly the same physical interface. On the other hand, server objects do not need to care about the location of client objects. They just provide an object interface which is exported through a skeleton. Thus proxies and skeletons completely hide the real communication process. Also, in most standard software middlewares the approach described above relies on the automatic generation of the proper proxies and skeletons depending on the kind of communication that must be established between objects.

We could consider the SoC just as another type of distributed system. Like such systems, a SoC is composed of a set of heterogeneous computing and storage resources linked through some interconnection infrastructure, and suffers the same kind of problems: scalability, heterogeneity, different communication technologies, ... Hence, it seems

reasonable to apply the same kind of solutions, and concepts, although not necessarily the same implementation.

In fact, the solutions to each scenario in section 2, completely match with the idea of RMI described above. The functionality of the adapters is the same than that provided by the proxies and skeletons. However, there are important implementation differences. One of the most important is, for example, that the middleware is normally built on top of several other SW layers, such as the Operating System. In our case there are not such layers, and the SoC middleware is implemented just on top of the System bus (note that we refer to a bus, but it may be indeed a hierarchy of buses or even networks inside the chip). The bus is already able to route the messages from one object to another, so there is no need of an extra layer for such purpose.

3.2 Proxies and Skeletons

The use and meaning of proxies and skeletons has already been shown in sections 2 and 3.1. However, from a more formal point of view, we could conclude that their mission is to provide transparency, where transparency has 3 different aspects:

1. In the location of the target, which is normally coded in the proxy, and not hard-coded in the object (the functionality)
2. In the implementation technology of the objects. Any proxy, whatever it is for a SW or a HW client will generate exactly the same transactions in the bus. That makes it impossible to know if the invocation came from a HW or SW object, as it also happens with the answer.
3. In the communication technology employed. We have mentioned in the example the OCP bus, but indeed we have implemented exactly the same examples using the OPB bus (see section 5) with the same results.

Finally, as also stated in the example we should highlight that proxies and skeletons can be generated automatically from the object interface description, so objects can be reused under any other different context just regenerating the corresponding adapters.

3.3 Hardware Cores

A hardware core in the SoC will be the combination of three parts: 1) the hardware object, which contains pure functionality; 2) one skeleton, as an adapter for those operations that the object is able to serve; 3) as many proxies as the object uses as a client.

From all three parts, only the object is meant to be reusable, while skeletons and proxies should be efficiently generated depending on each particular case.

But even those cores that have not been designed with this approach in mind may be used in the system middleware. For example, any RAM memory can be seen as an object providing read and write operation for bytes, words, double words, or even larger data blocks. The only thing required is to write a proxy that translates such operations into the proper transactions (DMA access for a block transfer, for example).

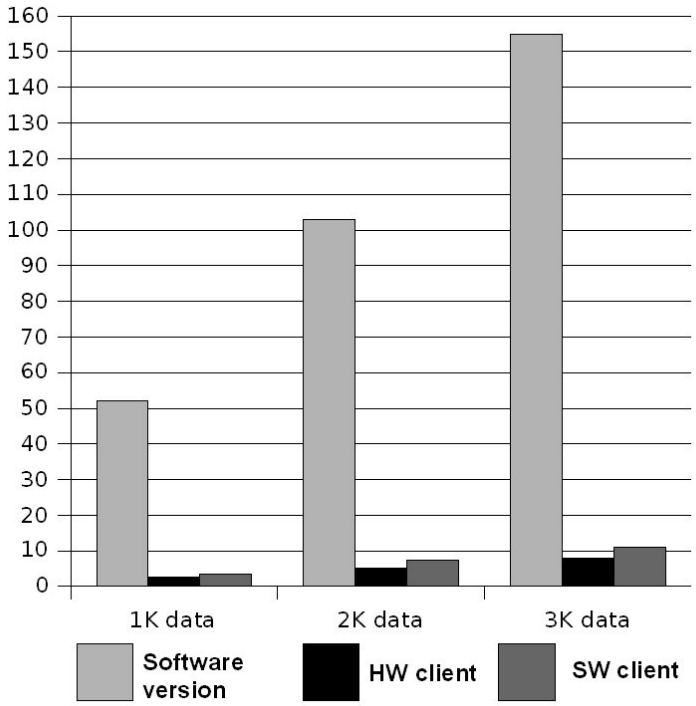

Figure 6: DES run times (in microseconds)

3.4 CPU Adapter

The main difference between hardware and software objects (in the middleware context) is that software objects share a common processing element, while hardware objects execute in their own. This makes it necessary some multiplexing mechanism for SW clients to have access to the bus. This multiplexer is called the Object Adapter, and consist in a set of SW routines with a standard API that must be linked with the object code of the SW clients. For every object to be able to have access to the bus, first it must be registered in an Object Adapter.

Another problem with HW to SW invocations is that objects inside the CPU are not visible out of it. CPUs are usually just masters of the bus, and are not addressable. Here the solution adopted has been to insert a bus interface between the CPU and the bus. In SW to HW invocations, the interface simply buffers the invocation and translates it into a bus transaction. In HW to SW invocations, the interface holds a translation table with bus addresses and object identities. If any of these addresses is detected in the bus, the interface buffers the transaction and notifies the Object Adapter in the CPU through an interruption. The OA then routes the invocation to the proper object, and the response back to the interface, if there is one. The interface then provides the server capabilities to the objects inside the CPU.

3.5 Remote Bridge

The aim of the remote bridge is to translate internal (to the SoC) invocations to external ones through some kind of network interface. The information that must be transmitted

on both sides of the communication has already been serialized, so the main task of the bridge is to pack it into the messages for a certain network protocol.

On the SoC bus side, the bridge listens for transactions addressed not to internal but external objects. Those are recognized through an internal translation table, where some internal addresses are mapped to the network addresses of the referred objects.

On the network interface it performs the same task, but in the opposite direction. But in any case, messages coming in and out of the interface have always exactly the same format than internal interactions.

4 Experimental Results

The example described in section 2 has been fully prototyped on the Xilinx XUP-V2Pro platform. First a completely SW version of the DES algorithm has been implemented on the Microblaze 32 bit processor, to use it as the software reference model. Next, all the middleware infrastructure has been generated for a SoC with HW, SW and remote clients for the DES model from Opencores.

Phases	**#cycles for 1x64 bit block data encryption**
Reception	218
bridge -> SoC bus	76
Execution	16
SoC bus-> bridge	72
transmission of the result	218

Table 1: Run times for remote execution encryption

Figure 6 shows the results for the DES encryption of a 2KB data block with the 55-bit key that the core supports. Each of the columns corresponds to a different scenario of execution. The first one is the fully SW version. The second one corresponds to a SW client that uses the HW DES to encrypt the data block. The third one is similar to the second, but this time we use a HW client. For the last two cases the DES core has not been modified at all. The results show how, as expected, the hardware DES implementation outperforms the software one. The small differences between the SW and HW clients are mostly due to the overhead introduced by the execution of the instructions in the software proxy.

In Table 1 we show the results for external invocation of the DES encryption though the ethernet device in the board. The run time is divided into five parts: the time for the reception of the packet, the time for the bridge to translate transactions into local messages, and the time to get the answer, and the time to build the response network packet. Clearly, the overhead of network transmission is very important with respect to the real execution time of the operation. However, the cost in area is paid only once (the cost of the bridge). Once included, any internal component (hardware or software) may be accessed from ouside the SoC.

	Part	*slices*	*slice FFs*
DES core	DES object	437	64
	DES skeleton	59	90
client core	DES proxy	65	96
	client object	82	125
Middleware	bus adapter	99	111
Middleware	remote bridge	606	633

Table 2: Area cost of the system

Finally, Table 2 shows the cost in area (slices inside the FPGA) of each of the components of the middleware plus the DES object.

5 Related Work

The ideas presented in this paper complement previous work on system-level abstractions. Orthogonalization of concerns in system-level design as proposed by [2], and more recently by [3] and [4], provide an object model similar to what this paper assumes, but most actual implementations focus on a structural view of the system and do not care about location transparency. In [5] a uniform communication mechanism for HW and SW resources is proposed, based on a central HW-SW OS and a HW abstraction layer to provide task abstractions for HW components. Previous works by Paulin et al. [6] already apply concepts from distributed object middlewares to SoCs but they do not even consider one of the key features, location transparency. Some early ideas on how reconfigurable computing may benefit from these concepts are found in [7]. Previous results on automated generation of communication infrastructure for SoC design in [8, 9] are also applicable to the proxy/skeleton generation.

Object-based and object-oriented approaches [10, 11] have also been used extensively to reduce the effort of translating some software components into hardware components or to improve the co-simulation of the system. Our hardware objects require a subset of what is provided by these extensions. Therefore we remain compatible with their approaches and we also keep full compatibility with standard IP based methodologies.

6 Conclusions

The communication architecture presented in this paper extends the distributed object paradigm to SoC platforms. The proxy and skeleton abstractions plus the use of the RMI semantics, provide a simple way to uncouple component functionality from communication implementation. From the designer perspective, this provides an homogeneous view of the system as a collection of communicating objects. From the implementation point of view, the model presented provides communication and location transparency for any kind of local interaction between hardware and software

components, blurring the hardware and software interface barrier. But it also provides the possibility of remote (may be off-chip) interaction with other objects.

Moreover, all the services and components that are part of the middleware can automatically be generated based on a few descriptions on the interfaces of the objects, and on the deployment over a certain platform. This enhances the possibility of future reuse and eases design space exploration tasks. And, as the experimental results show, the communication architecture does not incurs in high overheads.

Acknowledgments

This work has been funded by the Spanish Ministry of Education and Science under grant TIN2005-08719 and by the Regional Goverment of Castilla-La Mancha under grants PBI-05-0049 and PBC-05-0009.

References

[1] http://www.opencores.org

[2] Keutzer, K., Newton, A.R., Rabaey, J.M., Sangiovanni-Vincentelli, A. System-level design: orthogonalization of concerns and platform-based design. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, 19,* 12 (Dec. 2000).

[3] W. Ces rio, L. Gauthier, D. Lyonnard, G. Nicolescu, and A.A. Jerraya. Object-based hardware/software component interconnection model for interface design in system-on-a-chip circuits. The Journal of Systems and Software, (70), 2004.

[4] A. Gerstlauer, D. Shin, R. Dmer, and D. D. Gajski. System-level communication modeling for network-on-chip synthesis. In *Proceedings of theASP-DAC*, 2004.

[5] J-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *Proceedings of the DATE '03 Conference*, 2003.

[6] P.G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, O. Benny, D. Lyonnard, B. Lavigueur, and D. Lo. Distributed object models for multi-processor SoC's, with application to low-power multimedia wireless systems. In *Proceedings of the DATE '06 Conference*, Munich, Germany, 2006.

[7] Ronald Hecht, Stepah Kubish, Harald Michelsen, Elmar Zeeb, and Dirk Timmermann. A distributed object system approach for dynamic reconfiguration. In *Reconfigurable Architectures Workshop (RAW 06)*, Rhodos, Greece, April 2006.

[8] V. D'silva, S. Ramesh, and A Sowmya. Bridge over troubled wrappers: Automated interface synthesis. In *Proceedings of the Intl. Conf. on VLSI Design*, 2004.

[9] A. Gerstlauer. Communication abstractions for system-level design and synthesis. Technical Report CECS-TR-03-30, UC Irvine, 2003.

[10] Grimpe, E., Oppenheimer, F., Extending the SystemC Synthesis Subset by Object-Oriented Features. In *Proceedings of CODES+ISSS*, Oct. 2003.

[11] Schulz-Key, C., Winterholer, M., Schweizer, T., Kuhn, T., Rosenstiel, W. Object-Oriented Modeling and Synthesis of SystemC Specifications. In *Proceedings of theASP-DAC,* 2004.