Flexible, Open and Efficient Embedded Multimedia Systems

David de la Fuente, Jesús Barba, Fernando Rincón,
Julio Daniel Dondo and Juan Carlos López
School of Computer Engineering Department of Technology and Information Systems
University of Castilla-La Mancha, Ciudad Real
Spain

1. Introduction

Multimedia systems have to deal with the processing of large amount of data in a bounded time window. In order to fulfill the demanded requirements, the critical part of the system functionality has traditionally been implemented using custom hardware. The result is the improvement of both the throughput and the performance of the final system.

The first approaches dealing with hardware-based acceleration were based on the design of custom ASICs (Application Specific Integrated Circuits) or the use of DSPs (Digital Signal Processors). Lately, the emergence of high capacity reconfigurable devices such as FPGAs (Field Programmable Gate Arrays) is encouraging developers to use this technology for the implementation of multimedia embedded systems. Some of the reasons that back up this trend are: short development/prototyping time, the ability to customize hardware, its flexibility, or reconfiguration facilities, just to name a few of them.

Ideally, the development of a multimedia embedded system should be the result of a simple design process, where a set of cores are chained according to a certain execution flow, and later tuned to fit into a certain target platform. To achieve this scenario, a library of platform and vendor independent parameterized components is required. However, nowadays, this is an utopian vision since no standard with such characteristics is available. Each vendor develops their applications and components according to their own needs, without agreeing a common interface nor synchronization mechanisms with other third party actors. As a result, this implies extra difficulties when trying to port working solutions to other platforms. Therefore, a set of standards for developers is necessary in order to assure cross-platforms designs.

Currently, there are some standards for multimedia systems and applications development. However, vendors would rather not assume the cost derived from the adaptation of their products (legacy or new) in order to be compliant with these new norms.

The lack of standardization is the source of ad-hoc solutions whereas the adoption of standards would provide a portable, flexible and reusable answer to the problems that arise in multimedia embedded systems design, and the related applications (i.e. delays in the commercialization of new products, reduced productivity, increment of the development costs, etc.).

In this chapter, a Hw/Sw framework for SoC based embedded multimedia systems is described. In addition to the benefits inherited from the use of the FPGA technology, our solution is also based on the use of OpenMAX standard [1]. This allows building multimedia embedded systems in a reasonable time and, at the same time, being such systems fully reusable and portable.

OpenMAX is an open standard promoted by the Khronos Group [1] which pursuits to reduce the cost and complexity of porting multimedia software to new processors and architectures. Since the OpenMAX standard provides an only software reference implementation, it has been necessary to revisit and redesign all the architectural concepts and protocols of OpenMAX for the heterogeneous embedded systems ecosystem; this means that the proposed framework is intended for hybrid (Hw/Sw) multimedia systems development. Therefore, a flexible and transparent mechanism to manage Hw/Sw interfacing is necessary. Moreover, to cope with the demanded temporal and computational requirements, it is mandatory to keep the overload introduced by the integration infrastructure to a minimum.

The framework described in this work borrows the facilities provided by the Object Oriented Communication Engine (OOCE) [5]. OOCE is an heterogeneous middleware for SoCs. It provides basic and advanced in-chip communication services to transparently handle communication between the software and the hardware parts of an embedded system. In order to meet the performance levels demanded by multimedia systems, in this work we have extended OOCE with new features and some optimizations. The framework also follows a design approach based on MDA (*Model Driver Architecture*, [6]), in order to ease the design workflow (as it is stated in the "MDA manifest" [7]). The main purpose of the framework is then raising the level of abstraction when defining the multimedia system, in order to facilitate the automatization of the design, analysis and verification through the use of executables models.

2. Related works

Multimedia applications have their own requirements that differentiate such applications from other domains. For example: an intensive traffic of data and use of the memory subsystem, a characteristic application model, real time restrictions, etc. These particular features have led the research community to invest a significant amount of effort in projects addressing the specific challenges imposed by the development of multimedia (embedded) systems.

The most widespread solution to improve the throughput relates to to the development of the multimedia platform with hardware acceleration in mind. This solution is mentioned in [8] and [9], which respectively implement the computational cores as coprocessors or heterogeneous reconfigurable engines.

The Cell BroadBand Processor [10], with its characteristic four bus ring, combines a general-purpose Power Architecture core with streamlined coprocessing elements which greatly accelerate multimedia and vector processing applications.

Recently, many-core architectures for graphical applications, such as NVidia GPUs arrays, have gained a great importance mainly due to the availability of CUDA [11], an open software development framework based on a standard widespread language as it is C.

[1] `http://www.khronos.org/`

PeaCE [12] is another alternative to develop Hw/Sw multimedia platforms. PeaCE provides a development flow (from functional simulation to the synthesis process) to multimedia applications with real time constraints.

An important factor of these platforms is how the components in the system communicate. Generally, communication mechanisms are fixed, with a null or limited capability to be customized. Since there are many factors that can affect to the communication performance that should be considered, offering the ability to reorganize the communication channels in run time is essential. This would also be a useful tool for designers that could quickly and easily perform a design space exploration.

For example, Na Ra Yang et al. [13] propose double buffer, open row access and interleaved memory techniques in order to achieve an efficient communication. Another alternative to improve the communication performance is described in [14] which presents a solution based on a time multiplexed memory approach, where the system memory is accessed several times during a single bus cycle. To allow this, the memory and the DMA logic must be operated at a clock frequency which is a multiple of the clock frequency of the microprocessor.

In relationship to the performance analysis of communications in multimedia dedicated platform, in [15] several alternatives for a shared memory data exchange mechanism are compared: point-to-point connection, and based on bus architecture using DMA. In the the final part of the work, they present a performance estimation tool based on a Bus and Synchronization Event Graph (BSE graph) that is used to obtain statistical results.

Concerning the use of MDA techniques for multimedia embedded systems, MARTES (Model-based Approach to Real-Time Embedded Systems development, [16]) is a project whose main goal is the use of UML and SystemC in embedded systems. MARTES has taken some ideas from MDA, particularly, the separation of the application model from the target platform. Ying Wang et al [17] propose an MDA approach combining UML and SystemC in order to promote stepwise semiautomatic conversion from UML specification to executable SystemC code. They intend to build a smooth SoC design flow in which the implementation can be derived directly from the specification.

3. OpenMAX

The OpenMAX standard is an initiative promoted by the Khronos Group that aims to unify the way media components interoperate, in order to reduce the cost and complexity of porting multimedia software to new processors and architectures.

As shown in figure 1, OpenMAX is composed of three layers:

- OpenMAX Application Layer (AL, [2]) provides a standardized interface between an application and the multimedia middleware that provides the services needed to perform an expected API functionality. OpenMAX AL provides application portability with regard to the multimedia interface.
- The OpenMAX Integration Layer (IL, [3]) is an API that provides access to the software components in a system. In this layer, a standardized procedure is defined for the activation, initialization, creation and disposal of components.

  This layer contains three main entities (figure 2):

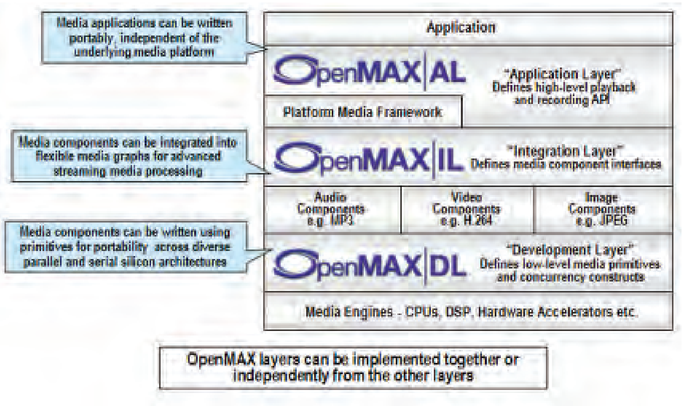

Media applications can be written portably, independent of the underlying media platform
Media components can be integrated into flexible media graphs for advanced streaming media processing
Media components can be written using primitives for portability across diverse parallel and serial silicon architectures
Application
OpenMAX AL
"Application Layer" Defines high-level playback and recording API
Platform Media Framework
OpenMAX IL
"Integration Layer" Defines media component interfaces
Audio Components e.g. MP3
Video Components e.g. H.264
Image Components e.g. JPEG
OpenMAX DL
"Development Layer" Defines low-level media primitives and concurrency constructs
Media Engines - CPUs, DSP, Hardware Accelerators etc.
OpenMAX layers can be implemented together or independently from the other layers

Fig. 1. OpenMAX Portability Library [1]

– The OpenMAX IL Client is the entity that interacts with IL Core. Normally it is a functional piece of a framework or application.
– The OpenMAX IL Core is used for dynamically loading and unloading components and facilitating component communication. Once loaded, the API allows the user to directly communicate with the component, which reduces the system overhead.
– The OpenMAX IL Components (OMX Components) represent individual blocks of functionality. Components can be sources, sinks, codecs, filters, splitters, mixers, or any other data operator.

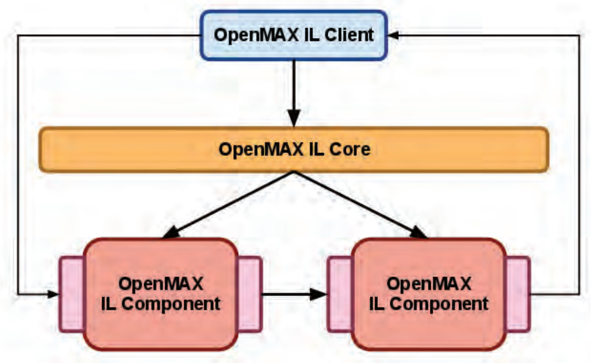


Fig. 2. OpenMAX IL structure

- The OpenMAX Development Layer (DL, [4]) defines a set of low-level multimedia kernels, or media processing building blocks, that might be used to accelerate traditional computational hotspots. The functional scope of the DL API spawns several key domains to mobile multimedia platforms, including the following: signal and image processing, audio coding, image coding, and video coding.

The typical multimedia application topology in OpenMAX depicts a chain of OMX Components that process the data flowing from their inputs to their outputs. OpenMAX offers a standardized component interface for each of the following domains: audio, video and image data.

OMX Components handle data communication through ports. A port is an interface that represents: the connection to other components, the type of streamed data and the *buffers* needed to maintain such connection. In OpenMAX a *buffer* is an entity that holds the information and represents the minimum unit of data that can be exchanged between two OMX Components. One OMX Component sends/receives buffers to/from other OMX Components through output/input ports.

3.1 OpenMAX communication models

Communication between components can take place in three different ways, as it is described in the standard. The designer is free to choose the one best suited to his needs. Therefore, the variety of communication models provided by OpenMAX makes the adaption of applications easier. Briefly, we summarize the three types of communication which are depicted in figure 3:

- *Non-Tunneled*: the communication between components is established through the entity that implements the application control. In figure 3 OMX Components A and B use this scheme.
- *Tunneled*: the communication between components takes place directly. To this end, the standard defines a buffer exchange convention. In figure 3 OMX Components B and C use this scheme.
- *Proprietary*: the communication between components takes place directly but, opposite to the tunneled case, the mechanism is not defined by the standard. In figure 3 OMX Components C and D use this scheme.

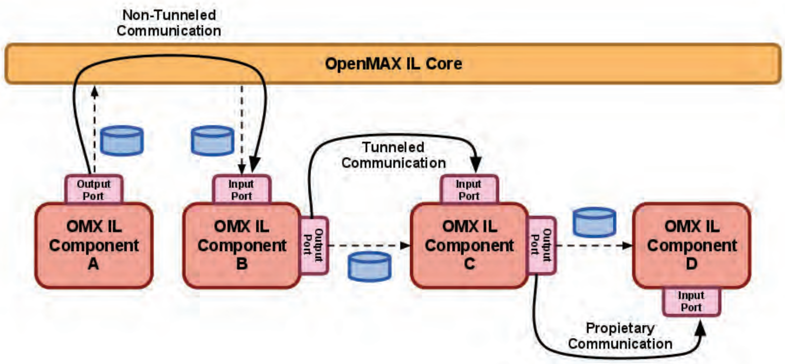


Fig. 3. OpenMAX communication models

3.2 OpenMAX in this project

Lately, the utilization of FPGAs to instantiate components for multimedia system acceleration has grown significantly. Through the use of OpenMAX in embbedded systems, developers can reduce the effort required to design multimedia hardware because: (a) the whole core

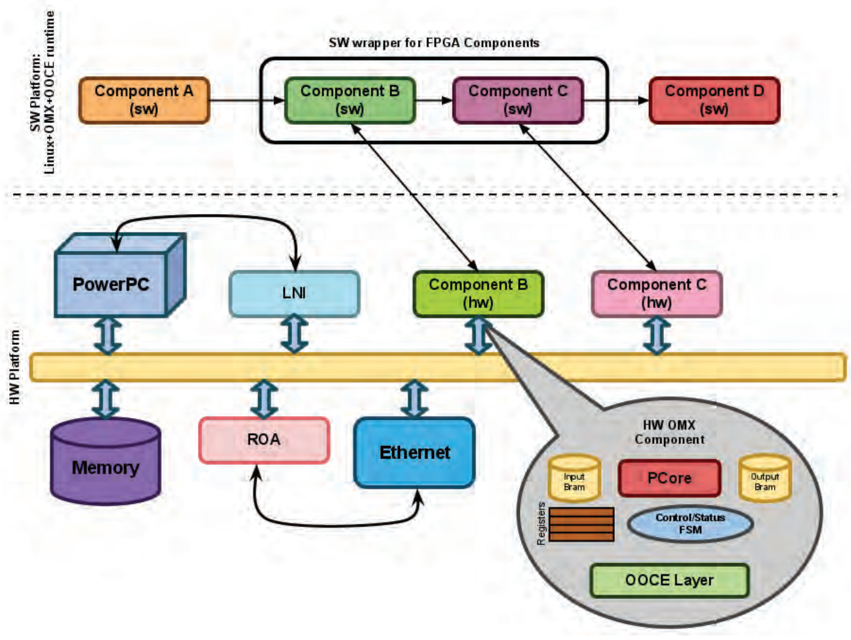


Fig. 4. Example of an implementation of an heterogeneous Hw/Sw multimedia processing chain with Hw OpenMAX Cores

logic can be reused when targeting a new platform (standardized interfaces), (b) it is not necessary to hand write the drivers or code that depends on such components and, (c) communication issues are separated from the processing primitives (same synchronization protocols, standardized communication mechanism, etc.).

However, the standard only specifies the primitives and services which conform the different API layers, always from an only-software point of view. So, the OpenMAX vision has been extended, mainly working at the Integration and Development layers of the standard, to embrace hardware components implementing accelerated multimedia functions.

In Figure 4 a representation of a heterogeneous multimedia processing chain is depicted. Some of the components are implemented, using our approach, as hardware accelerators. In this example, *components B and C* in the chain are mapped to a hardware implementation of an *OpenMAX Core*. Every Hw OMX Component in the FPGA fabric has a software wrapper that implements the *OpenMAX Component class interface* in order to maintain the compatibility with the standard. However, the software wrapper has no responsibility at all in the data transmission process between Hw OMX Components that is actually performed without any software intervention. Synchronization, control and data flow management is performed by the hardware thanks to the *Hardware-to-Hardware Tunneled* invocation semantics provided by the integration infrastructure, which is described in more detail in the next subsection. This has a positive impact in performance since the processor is not mediating in every single data transfer that takes place in the system.

4. Object Oriented Communication Engine

OOCE is a middleware designed to manage the communications in a SoC through message passing. OOCE defines a series of architectural elements that provide a set of basic communication services inspired in the *Distributed Object Paradigm* and the *Remote Method Invocation* (RMI, [18]) semantics. The Distributed Object Model is the basis for an homogeneous communication mechanism, realized through an infrastructure that hides the implementation details from the designer.

In OOCE, every element in the system is an object that can be implemented either in hardware or software. Those objects make use of a set of components and hardware/software artifacts to invoke a functionality from others.

The OOCE infrastructure was designed with these key features in mind:

- Support all communication scenarios that can take place in a SoC. This even includes communication with external objects (outside the chip) through a network interface.
- Provide transparency and a unified view of the communications in the system.
- Facilitate the programmability, making the embedded software development simpler and more productive because of the use of automatic generation tools.
- Be efficient, scalable and fault tolerant, keeping the integration infrastructure overload in minimum levels.
- Be independent from a specific communication protocol or data transport layer.

OOCE is based on the Client/Server communication model with several additions in order to implement RMI semantics. The number and type of the messages exchanged between clients and servers are defined in order to invoke certain functionality in the latter. A message stands for a invocation and embodies: all the necessary information to reach its destiny (header), the parameters and a callback address if necessary (i.e. to deliver the response back to the client). Then, sending a message means to perform one or several transactions over the communication media (e.g., bus writes) with the destination address built from the header information. This communication scheme is common for any communication scenario regardless of the nature (Hw or Sw) of the objects involved. In the case of Hw actors, special adapters to the bus (proxies and skeletons) implementing the FSM which controls this process are generated in an automatic way.

4.1 OOCE workflow

One of the strengths of using OOCE is the capability to design the whole system with a single model (the object model), valid for all the system components regardless their implementation technology. This feature, together with a set of tools that enable the automatic generation of the platform-dependent communication adapters, makes OOCE very practical from the designer's point of view. Hence, each developer must be only focused in what he is really good for (for example, a hardware developer is concerned in writing efficient VHDL code), whilst the responsible of application coding has only to write good algorithms, using any available resource as a regular object.

As it can be seen in figure 5, application experts make homogeneous use of any platform resource in the form of object interfaces. The OOCE middleware provides the developers with the corresponding interface compilers for every technology involved in the system.

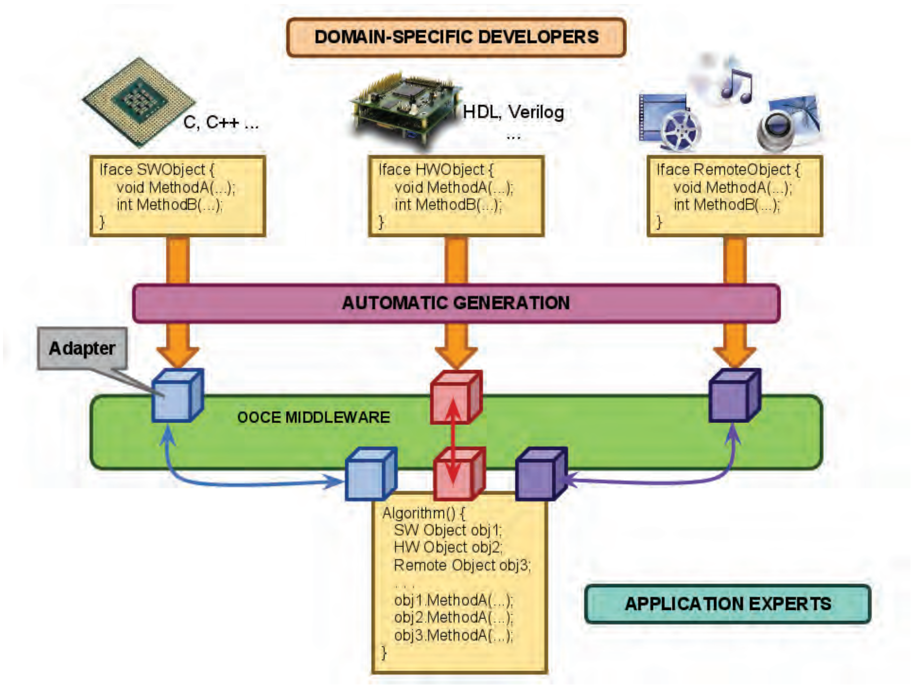
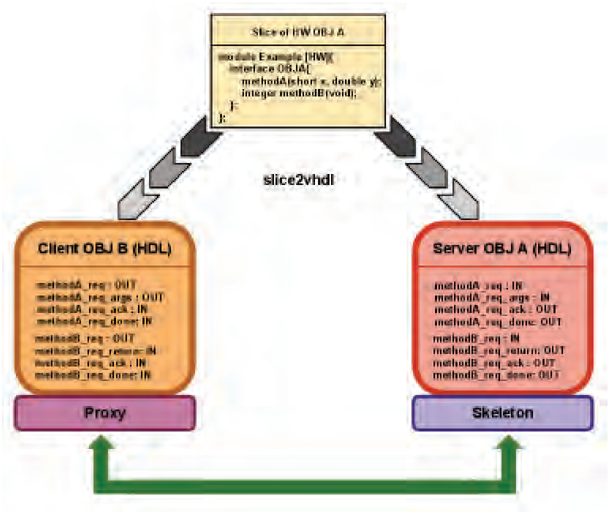

DOMAIN-SPECIFIC DEVELOPERS
C, C++ ...
HDL, Verilog ...
Iface SWObject { void MethodA(...); int MethodB(...); }
Iface HWObject { void MethodA(...); int MethodB(...); }
Iface RemoteObject { void MethodA(...); int MethodB(...); }
AUTOMATIC GENERATION
Adapter
OOCE MIDDLEWARE
Algorithm() { SW Object obj1; HW Object obj2; Remote Object obj3; . . . obj1.MethodA(...); obj2.MethodA(...); obj3.MethodA(...); }
APPLICATION EXPERTS

Fig. 5. Separation of roles, unified model and automatic generation in OOCE

Object interfaces are described using an Interface Description Language (IDL). In OOCE the IDL used to this end is Slice (Specification Language of ICE, the Internet Communication Engine, [19]). Once the interfaces have been declared, the interface compiler generates the adapters for each object (proxies and skeletons that are responsible for building/decoding the invocation/response messages). Depending on the nature of the object, the task can be "slice2vhdl" for hardware objects or "slice2<object_language>" for software objects. The generated adapters are optimized to keep the overhead to a minimum. Through this adapters and thanks to OOCE infrastructure each object can communicate in an homogeneous and transparent way with the rest of the object of the system.

As an example of what a hardware object for OOCE actually means, let see figure 6. Here, the entity interface for a hardware module and its skeleton (which connects to the bus) are shown, for example, and interprets the protocol and signals, redirecting the transactions that mean the activation of an operation on the hardware module. If another hardware object needs to execute some functionality from *OBJ A*, a skeleton module will be attached to it (see figure 6 on the right side). The skeleton is the mirror image of *OBJ A*, with the same entity interface. Thus, *OBJ B* will use the point-to-point invocation protocol whereas the actual invocation takes place through the bus. Since both of them are generated in an automatic way from the object interface description, the internal format used to code the messages is completely transparent to the user. Developers should only focus their efforts on implement the functionality of the object respecting its interface.



```
module Example [HW]{
  interface OBJA{
     methodA(short x, double y);
     integer methodB(void);
  };
};
```

Fig. 6. Automatically Generated Hw Object

4.2 OOCE communication models

As mentioned in the previous subsection, OOCE oversees every transaction between objects in the SoC. The actual methods implemented by the communication systems are characterized by the following features: (a) Hw objects interoperate transparently with the Sw subsystem and conversely; (b) efficient communication mechanisms between any kind of component and; (c) flexibility.

From the whole system point of view, the main advantage is the transparency in the communications between HW/SW objects with a low overhead. In order to achieve this, the system must support three types of communication scenarios: Hw/Hw, Hw/Sw and Sw/Hw. Apart from these three types, OOCE offers the possibility of establishing communication with objects that are outside the chip ("External communication"). Such objects can be regular ICE objects running on a computer or OOCE Hw/Sw objects in a OOCE-compliant SoC.

To establish any communication in the system implies that there must be at least one proxy for each client object that wants to make an invocation, and exactly one skeleton for each server object.

In figure 7 there is an example of a system composed by four OOCE objects: two clients ("Obj4" is a Sw object and "Obj2" is a Hw object) and two servers ("Obj1" in Hw and "Obj3" in Sw). Clients can require the functionality provided either by a Hw or Sw server through their proxies. There is also an external object that is connected with "Obj1" through the Remote Object Adapter (ROA). The three types of communication paths, plus external communication, are represented in the figure.

4.2.1 Hw/Hw communication

This type of communication occurs between two hardware objects that are connected to a common transport architecture. Within the SoC framework we focused on bus interfaces and

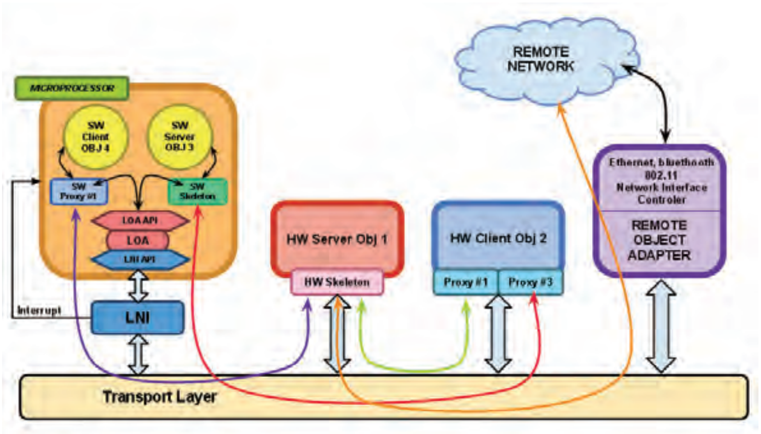


Fig. 7. Example of different communication types in OOCE

hierachies but this approach is flexible and easy to port to other paradigms such as Networks on Chip.

Back to Hw-Hw communication, the elements involved are: (a) A hardware client object, (b) the proxy of the client, (c) a hardware server object and (d) the skeleton of the server. The sequence of generic Hw/Hw invocation steps will be described:

1. The client makes the invocation, activating the corresponding signals of its object interface.
2. The proxy receives the invocation and its parameters and translates them into one or more transactions that are sent through the bus.
3. The skeleton recognizes the address of the transaction, so it reads the data contained in the bus transfer.
4. Finally the skeleton transforms the transaction into an invocation and sends it to the server through the server interface signals.

Figure 7 represents Hw to Hw communication between the client object "Obj2" and the server object "Obj1".

4.2.2 Hw/Sw and Sw/Hw communication

OOCE offers a unique sight of all system objects and a common semantic to their functionality access. In OOCE communication between Hw and Sw objects is done transparently. In order to establish a Hw/Sw communication it is only necessary to incorporate a minimal infrastructure, since the nature of the objects is different, that does not affect the former implementation of the Hw object nor the way the application uses such objects. The elements that have been incorporated to achieve this goal are: the Local Network Interface (LNI) and the Local Object Adapter(LOA).

In this case, the Hw client object invokes through the proxy which generates the corresponding bus transactions. Then, the LNI (which serves as a bridge to the Sw objects that are running in the processor) forwards the traffic whose address is linked to any of the Sw object references.

The LNI has a routing table in which there is information to map physical address (object identifier) to running software processes. The invocation and its parameters are passed to the LOA that finds the proxy/skeleton in charge to process them.

In the figure 7, both types of communication are represented between: (a) the client object "Obj2" and the server object "Obj3" for Hw/Sw communication and (b) the client object "Obj4" and the server object "Obj1" for Sw/Hw communication.

4.2.3 External communication

OOCE allows other entities (typically other OOCE objects in a different SoC or ICE Sw objects) to communicate with internal objects. This service is totally transparent to the internal objects. Again, transparency applied to this scenario means that objects and OOCE associated infrastructure should not change or adapt their behavior to this new scenario. For this reason, OOCE defines a subset of features that are 100% compatible with the ICE commercial middleware. The system data type and encoding rules of OOCE distributed object model are fully compatible with the data coding scheme of ICEP (ICE Protocol). This make it easier, to provide OOCE with the ability to communicate with ICE objects, because it is not necessary to re-encoded the body of the message when it travels from the in chip environment to the external network.

The Remote Object Adapter (ROA) is devoted to adapt the external and internal communication protocol.

4.3 OOCE extension

The need to extend OOCE appears when it is necessary to support the requirements of multimedia systems. Indeed, the first step was to identify the features of the OpenMAX standard that would mean changes or adjustments in the former OOCE architecture. Following, we enumerate the most important ones:

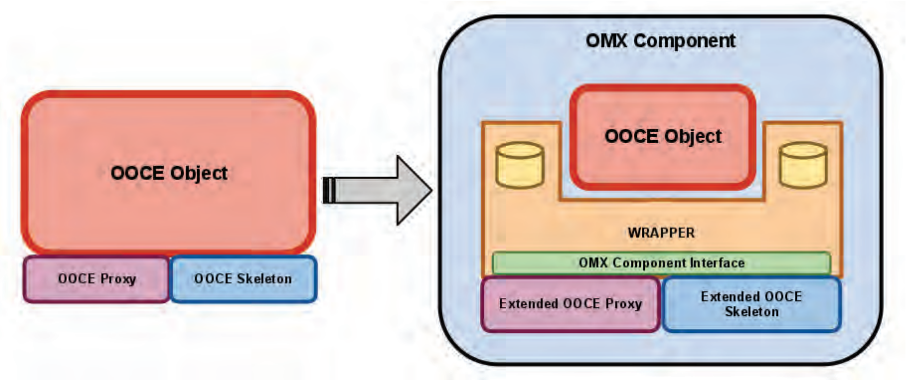


Fig. 8. From object model to component model

- The communication in OOCE (Hw/Hw, Hw/Sw and Sw/Hw) is message oriented, not intended for shared memory communication (data oriented). Therefore, OOCE must allow the components to exchange information through shared memory in a safe way.

- OpenMAX offers three types of communication mechanisms that should be implemented by OOCE, so both proxies and skeletons must be upgraded with new data types and control logic.
- OOCE objects should behave as OMX Components, so it is necessary to provide OOCE objects with a wrapper that implements the part of the OpenMAX Component's functionality that is relevant from a hardware implementation option (e.g., parameter initializaton).
- The logic to manage the communication and the object state must be added.
- The implementation of the OpenMAX methods should not affect the OOCE objects.

In addition, the overhead introduced by the above modifications should be as small as possible.

5. An OMX realization for FPGAs

The solution architecture that we propose is based on FPGA devices (in particular ML507 Virtex-5 Development System). In order to easily port the OpenMAX libraries to this board, an adapted version of a Linux Kernel (Linux-2.6-xlnx.git for XUPV5-LX110T) has been used as the operating system in our prototypes. This allowed us to have the multimedia application implemented entirely in software, as the starting point.

Then, the application components that are candidates for hardware acceleration have to be implemented as Hw OMX Component (described in detail next). At this point is where the extended OOCE comes in, providing the standard compliant Hw/Sw communication mechanism.

So, the bulk of this proposal is the implementation of all communication alternatives between two OMX Components. Communication between two OMX Components that remain implemented in Sw is not treated here since it follows the same principles and mechanism as defined in the standard, and the implementation reference provided by Khronos. So, we concentrate on those scenarios where a Hw version of the OMX Component is present. Every Hw OMX Component needs a software counterpart that interacts with the rest of the OpenMAX middleware and other software components. This is mandatory in order to keep the rest of the application unchanged. Such software wrapper stands for the hardware component and exactly implements the same API required by the standard. The difference is that this is a modified version of the Sw OMX Component, that is, a facade that only redirects to the Hw OMX Component invocations related to the hardware. As an example, the *SetupTunnel* primitive involving two Hw OMX Components is translated by our OMX Component (the software wrapper) into several OOCE invocations to configure the internal registers in the hardware. These registers indicate the base address where data must be written or read.

To illustrate what *transparency* and *easy integration of Hw components* actually mean, let us introduce the listing 1 where an extract of an OpenMAX application example running in a FPGA is represented. The application receives one video frame from the Ethernet interface, it transforms the RGB picture to grey scale, it applies a border detection algorithm and it finally sends the result via Ethernet. The *img_ethreader* and *image_ethsink* components are in charge of the acquisition of the image to be processed by the chain, and the delivery of the result, respectively. These components run in software, whereas the *hw.img_RGB2BW* and *hw.img_sobel* components are implemented in hardware. Nonetheless, there are no distinctions between the use of the hardware components and the software ones since both of them are OMX compatible.

Listing 1. Example of an implementation of an heterogeneous Hw/Sw multimedia processing chain with Hw OpenMAX Cores.

```
int main(int argc, char** argv) {
 /*Getting Components Handler*/
 OMX\_GetHandle(&appPriv->ethreader, ``omx.ext.image.ethreader'', NULL,
                &readercallbacks);
 OMX\_GetHandle(&appPriv->hwrgb2bw, ``omx.hw.img\_RBG2BW'', NULL,
                &RGB2BWcallbacks);
 OMX\_GetHandle(&appPriv->hwsobel, ``omx.hw.img\_sobel'', NULL,
                &hwsobelcallbacks);
 OMX\_GetHandle(&appPriv->ethsink, ``omx.ext.image.ethsink'', NULL,
                &sinkcallbacks);
 /*Set the size for img Hw OMX Components*/;
 sSize.sWidth.nValue = 640;
 sSize.sHeigth.nValue = 480;
 OMX\_SetConfig(&appPriv->hwrgb2bw, EXT\_OMX\_IndexConfigImgSize,
                &sSize);
 OMX\_SetConfig(&appPriv->hwsobel, EXT\_OMX\_IndexConfigImgSize,
                &sSize);
 /*Setting up tunneled communication*/
 OMX\_SetupTunnel(appPriv->hwrgb2bw, 1, appPriv->hwsobel,0);
 /* Change HW OMX Component state */
 OMX\_SendCommand(appPriv->hwrgb2bw, OMX\_CommandStateSet,
                  OMX\_StateIdle, NULL);
 OMX\_SendCommand(appPriv->hwsobel, OMX\_CommandStateSet,
                  OMX\_StateIdle, NULL);
 OMX\_SendCommand(appPriv->hwrgb2bw, OMX\_CommandStateSet,
                  OMX\_StateExecuting, NULL);
 OMX\_SendCommand(appPriv->hwsobel, OMX\_CommandStateSet,
                  OMX\_StateExecuting, NULL);
 ...
 OMX_DeInit();
 return 0;
}
```

Behind the scenes, there is much more to take into account for hardware components. In figure 9, a sequence of messages for a tunneled communication between Hw OMX Components is represented. The diagram shows how the OMX Component, that is used as a wrapper, extends the behavior of standardized messages by adding hardware invocations. For example, when a OMX Component receives the "start execution" by OMX_SendCommand(OMX_CommandStateSet, OMX_StateExecuting) notification, it sends an "active" invocation to Hw OMX Components.

Following in this section, the different communication models will be described, but before this it is important to define: *" What a Hw OMX Component is"*.

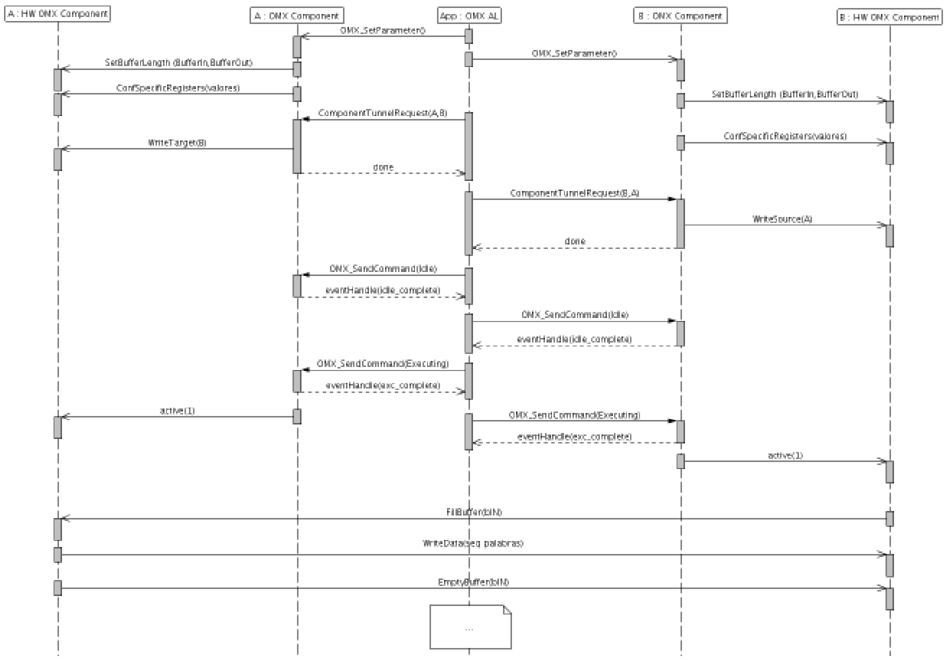

A : HW OMX Component
A : OMX Component
App : OMX AL
B : OMX Component
B : HW OMX Component
OMX_SetParameter()
OMX_SetParameter()
SetBufferLength (BufferIn,BufferOut)
ConfSpecificRegisters(valores)
SetBufferLength (BufferIn,BufferOut)
ComponentTunnelRequest(A,B)
ConfSpecificRegisters(valores)
WriteTarget(B)
done
ComponentTunnelRequest(B,A)
WriteSource(A)
done
OMX_SendCommand(Idle)
eventHandle(idle_complete)
OMX_SendCommand(Idle)
eventHandle(idle_complete)
OMX_SendCommand(Executing)
eventHandle(exc_complete)
active(1)
OMX_SendCommand(Executing)
eventHandle(exc_complete)
active(1)
FillBuffer(bIN)
WriteData(seq palabras)
EmptyBuffer(bIN)
...

Fig. 9. Init sequence diagram of tunneled communication between two Hw OMX Components

5.1 Hw OMX component

The effort of a multimedia embedded systems developer is too high. The OMX IL hardware adaptation we present in this work helps to reduce this effort enabling the designer to access to standardized procedures, automatic generation, etc.

Since the OMX IL is in the middle of the OMX software stack it bears some of the most important functionalities in OMX (initialization and connection of component as well as resource, communication and synchronization management). Therefore a Hw implementation of the main OMX IL Components will help to raise the throughput of an OMX based multimedia platform.

The Hw OpenMAX Component is the most important component in this layer since it embeds the hardware module that actually implements the multimedia function. That module, the inner one that appears in figure 10, is called *Processing Core* (PCore) and provides hardware acceleration to the corresponding multimedia function in the Development Layer.

As previously mentioned, each Hw OMX Component has an associated software wrapper. This wrapper principally acts as the facade to other OMX framework and application components, with the following goal:

- To act as a bridge for Hw/Sw interfacing operations. This includes, the configuration of the component, or the establishment of the Processing chain, for example. In this case, the standardized OMX Component class would have to be specialized and its methods would

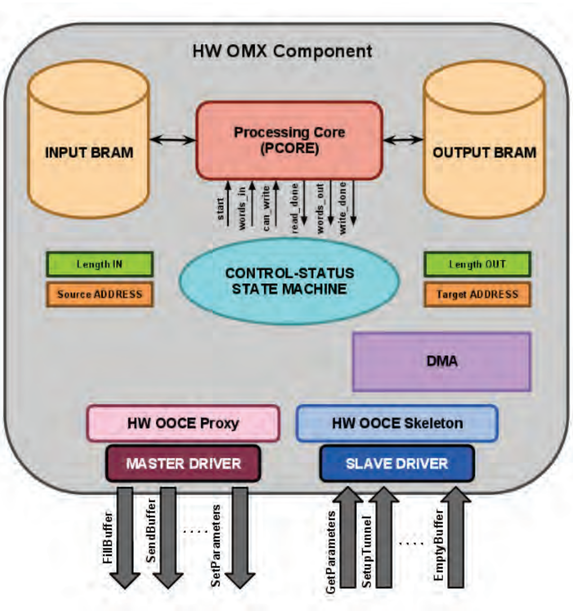


Fig. 10. Hw OMX Component structure

be rewritten to translate software invocations to input/output transactions over the bus through the OOCE middleware.

The OOCE skeleton detects and interprets OOCE messages that encapsulate the invocations addressed to the Hw OMX Component turning them into specific request to the internal units.

The architecture of the Hw OMX Component is thought as the placeholder for the PCore. Getting into details, the PCore has a fixed physical interface to the Hw OMX Component which makes it independent of the bus technology and, therefore, the platform in which it is going to be deployed. Figure 10 shows the structure of a Hw OMX Component mainly dominated by the presence of two local memories, where input and output data are stored. A complete buffer (the minimum data unit to be processed by a Hw OMX Component) must be in the input memory before any other action.

Then, an OMX "EmptyBuffer" operation needs to be received in order to "start" reading the input memory. Another signal, the "can_write" signal, is used to control whether it is possible or not to store data comming from the PCore in the output memory.

The number of data words that PCore has to consume is indicated by "words_in" signal. Once the buffer has been consumed (which means, completely read by the PCore and it activates "read_done" signal) a new one is demanded to the previous component (source_address register stores the address) in the chain ("FillBuffer" OMX operation). Notice that it is not necessary to wait for the Hw OMX Component to finish the processing of one buffer before asking for a new one. This optimization allows to parallelize component's execution with buffer transmission over the on-chip bus.

Once the component finishes the processing of the current buffer and writes the last word in the output memory (PCores activates the “write_done” signal), it transfers the output buffer to the next-in-the-chain component’s input memory (addressed by the content of the Target_address register) using a local DMA (Direct Memory Access) engine with a dedicated OOCE proxy. Now, the component is ready again to process the next buffer.

The Hw OMX Component implements two out of the three communication mechanisms described in the standard. Proprietary data exchange is not considered in this work since we believe the best benefit from an application is derived from the use of flexible and open models. As briefly sketched in the above paragraph, synchronization operations are handled separately (by the skeleton) from the data flow (by the internal DMA).

Although the Hw OMX Component can communicate in two different ways, we advocate the use of tunnel communication, which represents a decentralized model, because efficiency is an essential part in dealing with data transmissions in multimedia systems. This model is optimal in hardware since synchronization, control and data flow management are performed without intervention of the software, thanks to the Hardware-to-Hardware invocation semantics provided by the integration infrastructure (OOCE). It is easy to see the tremendous positive impact in performance this method has, since the processor is not mediating in every single data transfer that takes place in the system.

5.2 Hardware-to-Hardware communication model

Hardware-to-Hardware communication is based on the OMX Tunneled communication and is the most efficient communication type available in the standard.

OpenMAX defines a set of methods and primitives that must be observed when a buffer is exchanged between components. The communication meets the producer/consumer pattern and the components involved in the communication exchange “FillBuffer” and “EmptyBuffer” message types.

The goal is to transfer one buffer between two components without the intervention of any other component in the system. The sequence of steps in a Hardware-to-Hardware communication is the following:

1. The input local memory is empty. The component, as a consumer, sends a “FillBuffer” message to the producer component.
2. The consumer waits. The producer writes the buffer in the consumer’s input memory and sends a “EmptyBuffer” notification.
3. The consumer receives the “EmptyBuffer” invocation and begins to process the input buffer.
4. As the component processes the buffer, the resulting data are saved in the output memory.
5. When the consumer has read the last word from the buffer (not necessarily processed) it sends a “FillBuffer” notification to the producer again, avoiding unnecessary waiting.
6. When the output buffer is full and the consumer is ready to process The next buffer (“FillBuffer” message, step 1), the local DMA logic transfers the buffer.
7. Finally, the component playing the role of producer sends the “EmptyBuffer” notification.

In figure 11 a Hardware-to-Hardware Tunneled Communication between three Hw OMX Components is represented. The sequence of messages is also described in the picture. The

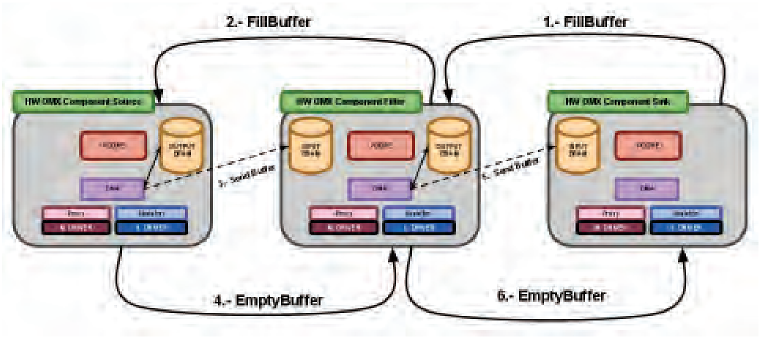


Fig. 11. Sequence of messages in Hw-to-Hw Tunneled Communication

first component in the chain is the *Source Component* which only produces buffers that are fed into the chain (a video streaming producer, ethernet reader or a simple file reader, as examples). The second is a *Filter Component* which plays the role of both a producer and a consumer. The last component in the chain receives the name of *Sink* which only consumes data (typically for storing, transmitting or displaying).

In this kind of systems, in which our work is interested, it is essential to get the maximum performance from the bus because it is the bottleneck in the communication. There is no discussion about the bad suitability of bus-based systems when the application bandwidth exceeds the bus capability. Nonetheless, the scenario we are considering is that where:

- Bus infrastructure is able to absorb the traffic generated by the multimedia application.
- Contention or arbitration issues and unoptimized transfers can result into failure.

Further optimizations have been developed in order to reduce the transmission latency of buffers through the bus and maximize the use of the available bandwidth. With the proposed optimizations, the number of cycles wasted due to punctual bus congestion is reduced, redistributing bus workload using techniques that parallelize and interleave all the transfers between OMX components. This is achieved by means of using the dead times while the component is processing a buffer.

The first optimization (*No Wait Until Fill*) configures the local DMA engine to transfer packets of N-words as soon as they are available in the output memory of the component. In the base configuration (*Wait Until Fill*), the whole output buffer must be in the output memory before the DMA starts transmitting it. The benefit of using this technique is twofold: (1) component's execution and output buffer transfer are overlapped; and (2) several buffer transmissions can take place at the same time (DMA pipelining). The figure 12 represents this optimization, in which the DMA engine sends data when the buffer is being filled without having to wait for its filling.

The second optimization (*multiple buffering*, figure 13) is independent of the DMA transmission configuration chosen. This technique allows to overlap: (1) in consumer mode, the reading of the actual input buffer with the writing of a new input buffer; and (2) in producer mode, the transmission of the actual output buffer with the writing of a new empty output buffer (Wait Until Fill configuration preferred). To this end, the physical address space of one memory is logically divided into N independently managed different regions. Thus, full parallelism is achieved.

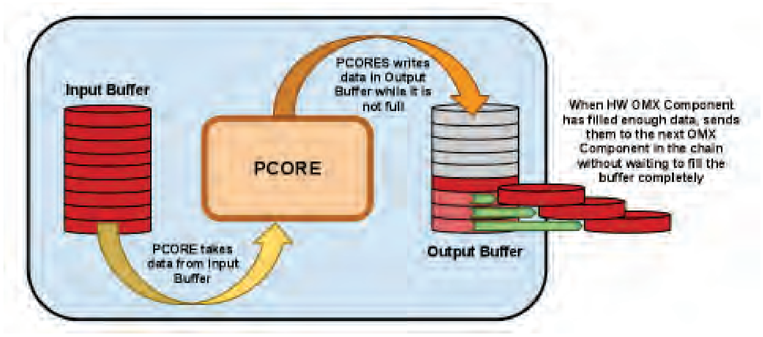

Input Buffer
PCORES writes data in Output Buffer while it is not full
PCORE
When HW OMX Component has filled enough data, sends them to the next OMX Component in the chain without waiting to fill the buffer completely
PCORE takes data from Input Buffer
Output Buffer

Fig. 12. *No Wait Until Fill* optimization example

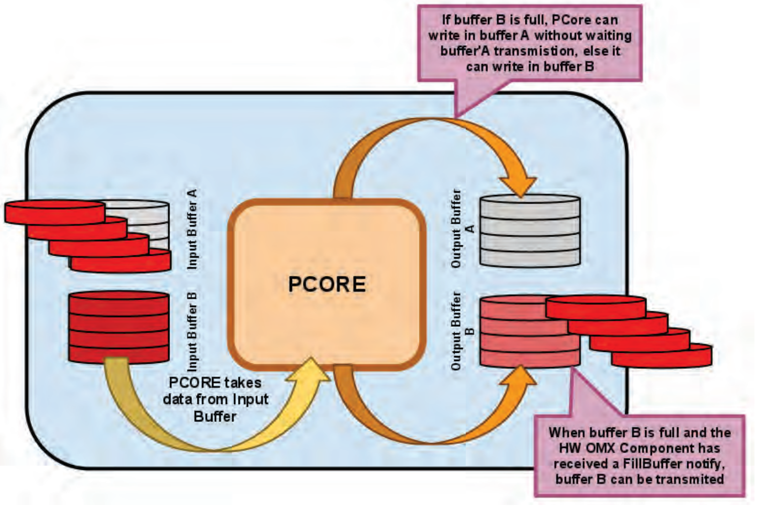

If buffer B is full, PCore can write in buffer A without waiting buffer'A transmistion, else it can write in buffer B
Input Buffer A
Input Buffer B
PCORE
Output Buffer A
Output Buffer B
PCORE takes data from Input Buffer
When buffer B is full and the HW OMX Component has received a FillBuffer notify, buffer B can be transmited

Fig. 13. Multiple Buffering optimization example

Both optimizations are compatible and they can be used simultaneously, but depending on the case their effectiveness might vary.

5.3 Hardware-to-Software communication model

Hardware-to-Software communication is also based in OMX Tunneled communication and it occurs between a Hw OMX Component and a OMX Component (software). This type of communication is not as efficient as Hardware-to-Hardware model, because of the unavoidable presence of software routines, with the associated increase in latency.

A Sw OMX Component reserves an area in memory to store buffers, so the exchanging of buffers takes place between the local memories in Hw OMX Components (Block RAM memory) and the DDR memory in our platform. In this model the DMA engine of a Hw OMX Component writes the buffer content into DDR memory and the synchronization messages are sent/received through the software wrapper used as the facade. It is worth mentioning that, from the Hw OMX Component implementation perspective, there is nothing to change since data transmissions are made through a proxy to memory. Thus, low level details related with the memory technology and the type of connection used are isolated from the the component's logic.

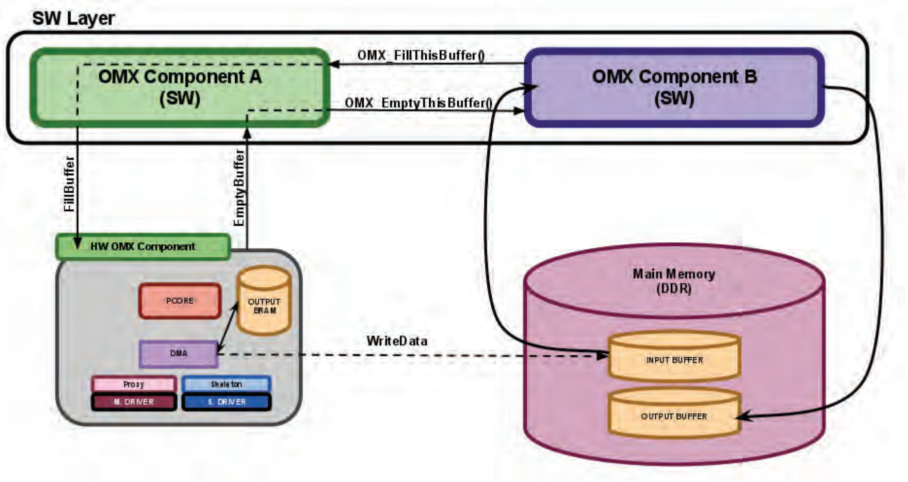


Fig. 14. Hardware-to-Software communication model.

In figure 14, the reader can see an example of Hardware-to-Software Communication model. As shown in the figure, the OMX Component A serves as a wrapper for the Hw OMX Component. This wrapper adapts standard methods into FillBuffer/EmptyBuffer invocations that will be sent or received from/to Hw OMX Component.

5.4 Software-to-Hardware communication model

In this case the Sw OMX Component acts as a producer and the Hw OMX Component as a consumer. The communication is based in the Tunneled OMX protocol too. In order to avoid two main memory accesses (one for writing the output buffer data and another for reading the input buffer data), the output buffer pointer of the Sw OMX Component is mapped to a region in the input local memory of the Hw OMX Component (see figure 15).

This type of communication has a problem in terms of efficiency because in Software-to-Hardware communication data is sent word by word, and burst transactions are not possible, so to use the input buffer of the Hw OMX Component as output buffer of the Sw OMX Component it is essential to get better communication performance.

With regard to the software wrapper for the Hw OMX Component, the implementation of the buffer destination pointer has been slightly changed. The Sw OMX Component still uses a memory reference to *a virtual output buffer* which really forwards memory writes to bus transfers that reach the BRAMs in the Hw OMX Component. This represents an advantage because there is only one copy of the buffer in the system, in contrast to the other models, requiring less memory resources.

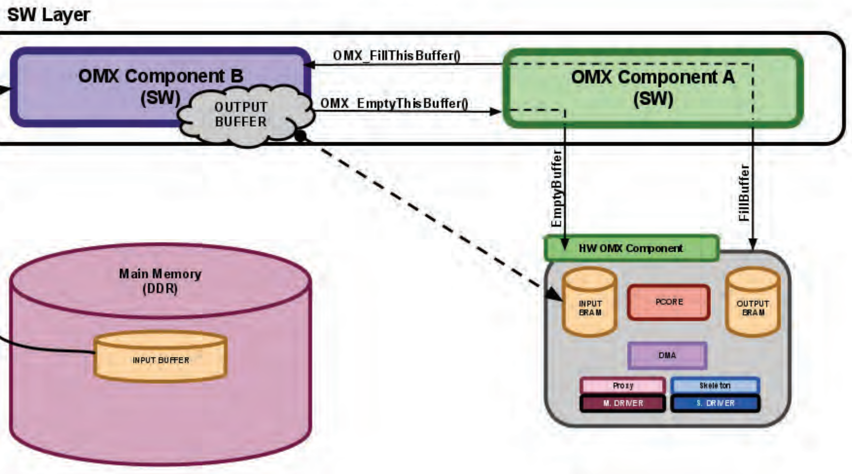


Fig. 15. Software-to-Hardware communication model.

5.5 Multimedia applications modeling for embedded systems using UML

This work aims to model multimedia embedded systems (based on OpenMAX standard) using some of the principles defined in Model Driven Architecture (MDA). For the correct application of MDA, a language that has a precise semantics in the system domain is necessary and it is being developed and is independent of any particular technology. For such a reason, the UML (Unified Modeling Language) has been chosen since it can be specialized or extended for dealing with specific domains or concerns. Moreover, the embedded systems domain needs to accurately express some domain-specific factors as synchronization mechanisms, memory capacity, power consumption or concurrency, for example. Because of this, we adopt MARTE (Modeling and Analysis of Real-Time and Embedded Systems) [20] as the basis to propose a UML profile intended to model embedded multimedia systems: MAME profile (Multimedia Application Modeling for Embedded systems).

MAME integrates OpenMAX in its semantics through the modeling of the main concepts. MAME profile is divided into two sub-profiles:

- *A hardware platform profile* to capture application-independent configuration of the SoC used to deploy the multimedia system. In our case, several prototyping boards from Xilinx have been modeled. This model feeds a set of scripts that generate automatically the infrastructure that enables the system implementation on a specific platform.
- *An OpenMAX profile* for modeling the multimedia applications based on our proposal for a OMX hardware realization. The definition drives the specification of the application through three different levels of abstraction.
  1. GAM Level (General Application Modeling) offers a general view of the whole application (see figure 16). This level reflects the different components that make up the application and concepts related to system architecture as: number of components, media data types, interconnection network, bitrate of the components, etc.
  2. IAM Level (Intermediate Application Modeling) provides a greater level of detail. In this level, the OMX Components are classified into hardware or software,

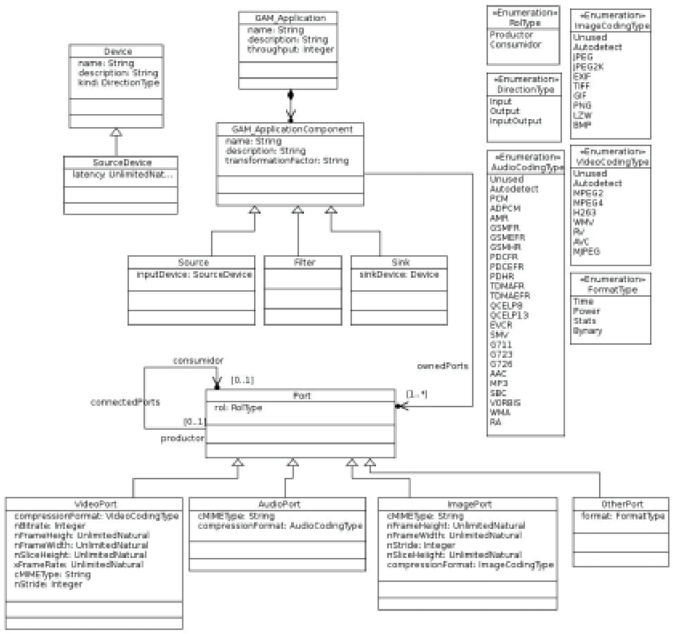


Fig. 16. GAM profile.

characterizing each one differently (see figure 17). The buffer concept is introduced together with the roles of each component.

3. CAM Level (Communication Application Modeling) incorporates the rest of the application attributes that have not been taken into account by the GAM and IAM profiles. It focuses on modeling the communication between the application components (see figure 18). The model supports the different scenarios of communication and optimizations described in previous sections.

5.5.1 Performance analysis of the application

As the result of the application of MDA techniques over both the platform and application models (figure 19), a Platform Specific Model(PSM) is obtained in order to be used to generate:

- An EDK-project with the instances of the necessary Hw OMX components.
- SystemC [21] simulation models that conforms an executable platform that allows testing the performance of the system.

There are two different types of analysis that can be done thanks to the SystemC models and MAME profiles above mentioned:

- Static. System bottlenecks can be detected or predicted through simple evaluations of the attribute values of the entities in the UML models using MAME. The goal is not to have accurate estimations but spot as soon as possible critical parts in the system architecture: memory bandwidth requirements, processing capacity, theoretical throughput, etc.

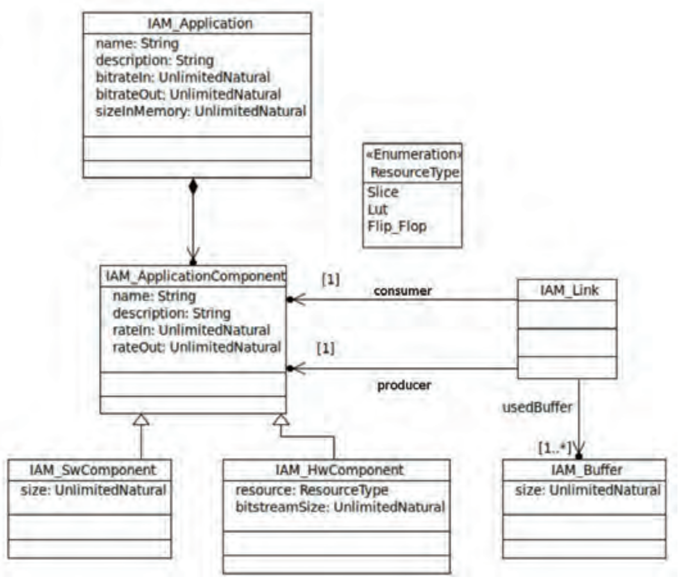


Fig. 17. IAM profile.

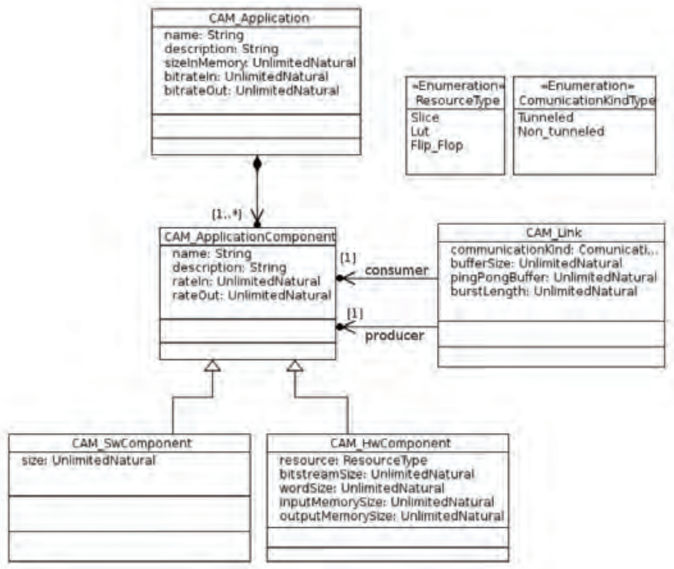

CAM_Application
name: String
description: String
sizeInMemory: UnlimitedNatural
bitrateIn: UnlimitedNatural
bitrateOut: UnlimitedNatural
«Enumeration»
ResourceType
Slice
Lut
Flip_Flop
«Enumeration»
ComunicationKindType
Tunneled
Non_tunneled
[1..*]
CAM_ApplicationComponent
name: String
description: String
rateIn: UnlimitedNatural
rateOut: UnlimitedNatural
[1]
consumer
[1]
producer
CAM_Link
communicationKind: Comunicati...
bufferSize: UnlimitedNatural
pingPongBuffer: UnlimitedNatural
burstLength: UnlimitedNatural
CAM_SwComponent
size: UnlimitedNatural
CAM_HwComponent
resource: ResourceType
bitstreamSize: UnlimitedNatural
wordSize: UnlimitedNatural
inputMemorySize: UnlimitedNatural
outputMemorySize: UnlimitedNatural

Fig. 18. CAM profile.

- Dynamic. Running the SystemC behavioral models, the developer gets more accurate timed information about the execution of the application. This simulation model allows the user to simulate the behavior of the entire system, offering a flexible mean to play with some component or system configuration parameters (i.e. size of the buffers) and quickly obtain data to:
  - Measure the impact of the application of optimization techniques.
  - Help the designer to explore the design space more efficiently in order to get the maximum performance.
  - Know the average time spent by each component processing or transmitting data and waiting for the bus.
  - Avoid configurations that would lead to bus congestion.
  - Maximize the use of the communication infrastructure and at the same time minimizing the dead times components would be wasting, waiting to be granted by the bus.

  After this analysis, the designer can make the appropriate changes and checks the results running the simulation again.

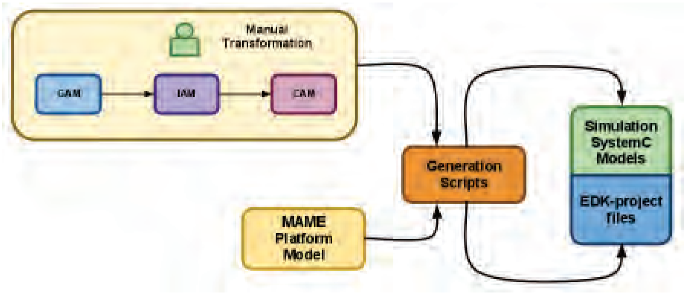


Fig. 19. Automatic generation from MDA application.

6. Experimental results

In order to provide the reader with a glimpse of the benefits and efficiency of the proposed approach, two experiments are dissected in this section. Particularly, we have focused in the analysis of the behaviour "No Wait Until Full" and "Multiple buffering" techniques and how they help to increase the system performance.

For all the experiments, there is a base configuration that comprises:

- 100Mhz clock system.
- 64 bits PLB bus SystemC model provided by GreenSoCs [22].
- 1Kword of local memories for all components.
- A 1Mword input file. The simulations consist in feeding the processing chain (variable components) with synthetic data comming from a file.

For different configurations of the processing chain and parameter values, several simulations have been carried out. It is worth mentioning that in this case we used PLB, but it is almost straightforward to perform new estimations over other platforms just changing the TLM model for the bus.

The first picture (figure 20) attempts to illustrate the benefits obtained by applying the "No Wait Until Full" optimization mechanism (described in section 5.2). As it is expected, the occupancy of the bus increases with the number of components in the processing chain. What is remarkable is how our proposal scales in a linear pattern. The measured transmission times are reduced in 20% (mean value) for the six cases when compared with the "Wait Until Full" strategy.

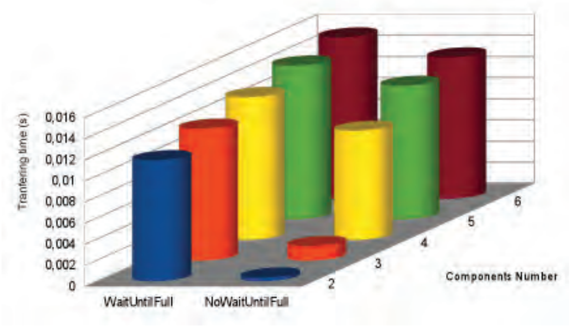


Fig. 20. Comparative chart of transferring time based on the number of components and "No Wait Until Full" optimization.

The second picture (figure 21) represents how the size of the buffer has an influence on the total transmission time. The benefits of the "Multiple buffering" technique are exposed in this experiments and its applicability is delimited, identifying such scenarios where its application makes sense.

Since the size of the local memories is 1Kword, the number of buffers that can be held in it varies (for example, two buffers for a 512 words buffer-size configuration). In this chart, we can see what is the best value for a certain buffer size configuration. To increase the number of buffers means that the control logic also increases, so it is necessary to estimate the cost/performance relation.

From the picture, two lesson can be learnt: (a) in systems with a low overhead regarding the bus bandwidth requirements, the bigger the size of the buffer the better the performance of

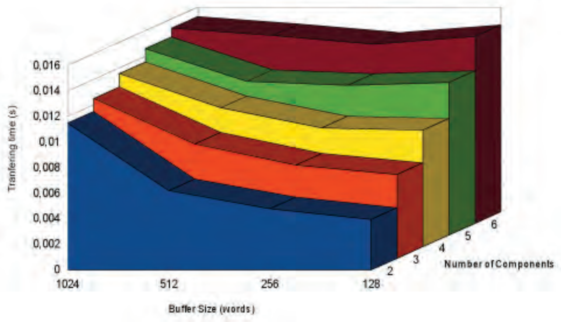


Fig. 21. Comparative chart of transferring time based on the number of components and "Multiple buffering" technique.

the system; whereas (b) in the opposite situation it is better to reduce the size of the buffer since several buffers transmissions can be interleaved, reducing the wait times.

7. Acknowledgment

This research was supported by the Spanish Ministry of Science and Innovation under the project DAMA (TEC2008-06553/TEC), and by the Spanish Ministry of Industry and the Centre for the Development of Industrial Technology under the project ENERGOS (CEN-20091048).

8. References

[1] The Khronos Group.http://www.khronos.org/openmax/

[2] OpenMAX Application Layer Application Programming Interface Specification. Version 1.0. 2007. The Khronos Group Inc.

[3] OpenMAX Integration Layer Application Programming Interface Specification. Version 1.1.1. 2007. The Khronos Group Inc.

[4] OpenMAX Development Layer API Specification. Version 1.0.2. 2005-07. The Khronos Group Inc.

[5] J. Barba, F. Rincon, F. Moya, J.D. Dondo, F.J. Villanueva, D. Villa and J.C. Lopez, *"OOCE: Object-Oriented Communication Engine for SoC Design"* DSD - Euromicro Conference on Digital System Design. Lubeck (Germany), 2007.

[6] Anneke Kleppe, Jos Warmer and Wim Bast. *"MDA. Explained The Model Driven Architecture: Practise and Promise"*. Addison Wesley, 2003

[7] G.Booch et al. An mda manifest. In D. and J. Parodi, editors, The MDA Journal: *"Model Driven Architecture Straight from the Masters"*, pages 133-143. Meghan-Kiffer Press, 2004.

[8] Martijn J. Rutten, Jos T.J. van Eijndhoven, Evert-Jan D.Pol, Egbert G.T. Jaspers, Pieter Van der Wolf, Om Prakash Gangwal and Adwin Timmer. *"Eclipse: A Heterogeneous Multiprocessor Architercture For Flexible Processing"*. Philips Research Laboratories, 2002.

[9] Paul Brelet, Arnaud Grasset, Philippe Bonnot, Frank Ieromnimon and Dimitrios Kritharidis. *"System Level Design for Embedded Reconfigurable Systems using MORPHEUS platform"*. IEEE Annual Symposium on VLSI, 2010.

[10] Kahle J. A., Day M.N., Hofstee H. P., Johns C. R., Maeurer T. R. and Shippy D. *"Introduction to the Cell Multiprocessor"*. IBM Journal of Research and Development. 2005.

[11] Buck, Ian. *"GPU computing with NVIDIA CUDA"*. ACM SIGGRAPH'07. 2007. San Diego (California).

[12] Soonhoi Ha, Sungchan Kim, Youngming Yi, Seongnam Kwon and Young-pyo Joo.*"PeaCE: A Hardware-Software Codesign Environment for Multimedia Embedded Systems"*. ACM Transactions on Design Automation of Electronic Systems, August 2007.

[13] Na Ra Yang, Gilsang Yoon, Jeonghwan Lee, Intae Hwang, Cheol Hong Kim, Sung Woo Chung, Jong Myon Kim, *"Improving the System-on-a-Chip Performance for Mobile Systems by Using Efficient Bus Interface"*. WRI International Conference on Communications and Mobile Computing, 2009.

[14] C. Brunelli, F. Garzia, C. Giliberto, and J. Nurmi, *"A dedicated DMA logic addressing a time multiplexed memory to reduce the effects of the system bus bottleneck"*, in Proc. Field Programmable Logic and Applications'08, 2008.

[15] Lahiri, Kanishka and Raghunathan, Anand and Dey Sujit, *"Fast performance analysis of bus-based system-on-chip communication architectures"*. Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design (ICCAD '99)

[16] Model-Based Approach for Real-Time Embedded Systems development project (MARTES), 2007. http://www.martes-itea.org/
[17] Ying Wang, Xue-Gong Zhou, Bo Zhou, Liang Lianga and Cheng-Lian Peng. *A MDA based SoC Modeling Approach using UML and SystemC*. Computer and Information Technology, CIT '06, 2006.
[18] Remote Method Invocation Home. http://www.oracle.com/
[19] The Internet Communications Engine. http://www.zeroc.com/ice.html
[20] The UML profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems http://www.omgmarte.org/
[21] Open SystemC Initiative (OSCI). http://www.systemc.org/
[22] The GreenSoCs website. http://www.greensocs.com/