A Hardware-Software Operating System: Using Heterogeneous Resources through Common Abstractions

José M. Moya*, Francisco Moya†, Fernando Rincón†, Juan Carlos López†
*E.T.S.I. de Telecomunicación
Technical University of Madrid
Email: josem@die.upm.es
†Escuela Superior de Informática
University of Castilla–La Mancha
Email: {Francisco.Moya,Fernando.Rincon,JuanCarlos.Lopez}@uclm.es

Abstract*— *Current embedded systems are made of multiple heterogeneous devices interconnected. These devices present a great variation of functionality, performance, and interfaces. Therefore, it is difficult to build applications for these platforms.

In this paper we present the notion of hardware-software operating systems to increase the abstraction level and to introduce component-based methodologies into hardware-software codesign. We make special emphasis on the use of simple, homogeneous interfaces to hide the inherent complexity of current designs.

We show, with a detailed example, that the use of a hardware-software operating system does not imply a significant overhead.

I. Introduction

As complexity increases, designing embedded systems is becoming too hard. A typical embedded system consist of an heterogeneous network with many different devices. These devices usually have very different interfaces (analog subsystems, microprocessors, field programmable devices, ASICs, etc.). The technology of the links between the devices can also be quite different, ranging from real-time network interfaces to dedicated buses.

The difficulties in developing applications for this kind of environments come from some limitations of current design methodologies. First, heterogeneity is handled with heterogeneity. We need different tool chains and different skills to design analog subsystems, microprocessor-based systems, digital signal processing units, etc.; also, complex hardware blocks usually provide equally complex interfaces. Second, the control of the application and the use of resources tend to be too centralized. A memory block is only used from its subsystem; common hardware-software partitioning techniques isolate the different subsystems and do not allow to share resources.

IP-based design strategies promise a significant reduction of the design time, but they do not solve the above problems. In fact, the integration of different IP blocks is becoming the key problem. This confirms that heterogeneity is bad and should be eliminated.

We are working on a complete design system to overcome all these limitations. Our main requirements are:

1) We are looking for a way to design really complex, distributed, hardware-software embedded systems, fast enough to satisfy the market demand.
   We want to be able to describe the system components at a very high level of abstraction, but we also want to be able to access to low-level implementation details when we need to. Thus, we have to support multiple abstraction levels at the same time.
2) The design environment should maximize reuse opportunities. We have to provide means to help the designer to reuse components from previous designs. We want to provide a component model with homogeneous interfaces so that any component in a design can be replaced easily.
   Also, to simplify the hard work of designing for reuse, the interfaces should be kept as simple as possible.
3) The overhead introduced by the higher abstraction level should be minimum.

To the best of our knowledge, no design system meets these requirements. Therefore, we are working on FLECOS [4] to build such a system. Its design relies heavily on simplicity and flexibility, as we show below.

In this paper we concentrate on the "operating system layer" of our design system, which is probably the part that best contributes to fulfill the requirements above.

In the next section, we give an overview of the FLECOS Operating System and how complexity can be reduced offering simple, uniform interfaces. In section III, we further describe some details of the implementation, illustrated with a detailed example, and we show some preliminary results. In section IV, we review some previous works that have influenced the design of FLECOS and its operating system. Finally, we briefly discuss some conclusions and propose future developments in section V and VI.

II. A Unified Approach to HW-SW Codesign

One of the reasons why component-based software is so popular, is that we do not need to learn a million of interfaces

to potentially use a million of devices. As long as your components adhere to a flexible, well-defined interface, they can be used in your system. Document models, such as OLE, DOM, or Bonobo, are examples of the power of this approach.

We want to apply a similar approach to the design of complex heterogeneous systems. Thus, we need to define what a component will be, how to define new components, and how to use these components to make a whole embedded system.

A first, intuitive, idea would be that a component is an abstraction of the resources it contains. This leads us to the definition of what we call *hardware-software operating systems*.

A. Hardware-Software Operating System

An operating system is, as given by Tanenbaum [10], "the software that securely abstracts and multiplexes physical resources". This abstraction plays a key role because

- it makes it easier and faster to design new applications,
- it keeps the application specification independent of the available resources,
- and it increases reuse opportunities.

All these characteristics clearly help us to fulfill the requirements given in section I.

Thus, we have slightly relaxed the definition so that it can be useful for hardware-software systems:

A hardware-software operating system is something that securely abstracts and multiplexes physical resources.

Those abstractions provided by the operating system will be used from the application through a well-defined interface. Therefore, we have to define this interface.

As we have seen before, we want to see an embedded system as a set of components, that can be modified or replaced freely. Each component is a set of resources (hardware, software, or a mixture) that implements the interface defined by the operating system. For example, in a Unix system, devices, sockets, directories, and regular files, are very different, but they all are files because they implement the file interface (open, read, write, close). Similarly, we want to use analog devices, ASICs, and software components through a common interface.

Usually, an operating system also provides some services, such as multiprocessing support, interprocess communication, memory management, etc. However, we do not want to pay for unused services. Thus, our hardware-software operating system does not provide any service itself. It only exports the available resources so that any required service can be implemented on top. This is similar to some modern microkernels, such as the exokernel [3], or Off++ [1].

B. The Box Interface

Basically, our prototype hardware-software operating system follows the model of the Off++ microkernel [1]. It only provides one abstraction for all the resources: the box. Everything is a box for the operating system: an analog filter, an A/D converter, a compression algorithm, a hardware multiplier, etc. The whole system can be seen as a box that contains other boxes.

A box can be formally defined as a set of resources (hardware and/or software) implementing the box interface. A box has one input and one output[1], and the interface is composed of only three simple operations:

`copy` Copies data from the output of the source box to the input of the destination box.

`select` Allows to access other boxes inside a box. The system is a hierarchical composition of boxes, representing different levels of detail. And the designer can always access to the system functionality at the required level, by going deeper into the hierarchy.

`bind` Inserts a box into another box attaching a new name to it. This operation allows to dynamically compose complex boxes based on simpler existent ones.

The box, as defined above, is just a set of requirements for every component in the system, but it leaves some details unspecified. Specifically, to be able to instantiate a real box in the system, the designer has to define the characteristics of the communication channels: the physical link, the signal types, and the communication protocols. This is similar to the C++ template mechanism. The box interface is like a C++ template: it is not a real type and we can not instantiate "pure boxes". But we can instantiate *concrete boxes*, with clearly defined input and output parameters.

Depending on the specific parameters of each box, we get different *types* of boxes. For `copy` to work properly, we need similar parameters in both boxes, both types should be compatible. The result is a strict hardware-software type system, with static type checking, and no extra overhead for hardware boxes.

The copy operation: For software boxes, a copy operation

```
copy (src, dest);
```

is translated into one or more data transfer instructions. The only requirement for two software boxes to be compatible for copy, is that there should be an agreement on the data size and type to be transferred.

But, what does copy mean for hardware boxes?

First, there is a data transfer from the output of the source box and the input of the destination box. Thus, there should be a link between both interconnection points, and there should be an agreement on the data types and the protocols. It may be needed to add some interconnection resources, such as analog and digital multiplexers, or buses, because there may be other copy operations using the same boxes within the system specification.

Second, the copy operation occurs at a time; it means "at this moment the input of the destination box has the same data as the output of the source box". Thus, it may be required to

[1] In Off++ there is a single input/output channel, which is more like a real box, but we have decided to clearly separate input and output to allow more flexible and efficient compositions. You can always connect both input and output to a single bus to resemble the real Off++ behavior.

add some storage resources to the input of the destination box to satisfy the copy semantics.

When the control unit issues a copy operation, the link should be activated (probably activating some control lines of a multiplexer), and the destination box should acquire the data. This justifies the need of a new control signal *copy(src,dest)*.

A copy operation between a hardware box and a software box is not possible, because the technology of the links is not the same. However, we can create a new box containing the software box, and including a hardware interface which is compatible with the hardware box. This is what we call *type converters* and we shall describe them in more detail below.

Copy-compatibility is not reflexive.

The select operation: A select operation

```
src->select (name);
```

returns a inner box of the source box, based on the specified selector. The selector type and size are characteristics of the source box type. There may be multiple selector types.

But, what does select mean in hardware boxes?

A hardware select is similar to a share operation with an internal *selected box*. The selector chooses between the possible selectable boxes and connects its input and output ports just like a share operation.

Of course, a box may have boxes of different types, with communication points of different and incompatible technologies. Thus, we need an internal *selected box* for every possible technology of the selectable boxes (i. e. analog and digital boxes). Note however that an 8-bit data bus may be connected to a 16-bit bus.

When the control unit issues a select operation, the links should be activated (probably activating some control lines of a multiplexer). This justifies the need of a new control signal *select(src,selector)*.

It is important to note that no additional hardware is required if select is not used. And, usually, we only use select for a few boxes in the system. Thus, the required hardware may be drastically reduced.

The bind operation: A bind operation

```
dest->bind (src, name);
```

makes the source box available from the destination box with the specified name. A box can be seen as a namespace and the `bind` operation creates or modifies the meaning of a name in that namespace. The new name can be used for other operations to refer to the original source box.

But, what does bind mean in hardware boxes?

First, there is no need to duplicate resources. A single box is connected to both inputs and both outputs. Of course, the technology of both communication points should be compatible.

Second, a bind operation starts at a time and lasts until another bind operation with a different source is issued by the control unit. It means "from now on, the input of the new box is the same point as the input of the source box, and the output of the new box is the same point as the output of the source box". Thus, it may be needed to add some interconnection resources, because there may be other bind operations with the same destination name.

When the control unit issues a bind operation, the links should be activated (probably activating some control lines of a multiplexer). This justifies the need of a new control signal *dest-¿bind(src,name)*.

A bind operation between a hardware box and a software box is not possible because of similar reasons to the copy operation, but we can use type-converters to achieve an equivalent behavior, as shown below.

Unlike copy-compatibility, bind-compatibility is reflexive.

There is no relation between copy-compatibility and bind-compatibility. One does not imply the other in any case.

C. Type Conversion

We have seen that every box in the system has a specific type, which determines the way it communicates with the external world. These strict types and the compatibility requirements of copy and bind help to catch errors early in the design process. However, to increase reuse opportunities, there should be a way to convert a type to another.

Therefore, we provide *type converters* that work as interface adaptors. A type converter is a user-defined box, containing the original box, but with the external interface of the destination type.

When used from an object oriented programming language, such as C++, we can use the implicit type conversion mechanism of the language to hide these details from the designer's point of view.

D. Using a HW-SW Operating System

The hardware-software operating system that we have described consists basically in three system calls (copy, bind, and select), that should be implemented for every contrete type of boxes (hardware or software).

In the system specification there nothing new with respect to a software-only operating system. Hardware resources are used through the system calls of the box interface. The low-level synthesis tools should generate the appropriate signals.

Section III-D gives some ideas on how we integrate the hardware-software operating system in out FLECOS design system.

III. An Example

As an example of the use of our hardware-software operating system, consider the system depicted in figure 1. It is an audio transmission system which uses MPEG-1 Layer 1 to encode the audio signal.

The figure shows that a box may be decomposed hierarchically into smaller boxes. This *divide-and-conquer* approach can be very effective to reduce the complexity of the problem, while maintaining a high level of abstraction.

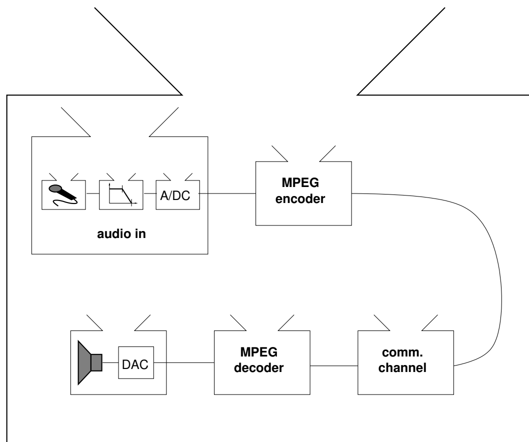

Fig. 1: MPEG audio encoder/decoder.

A. Architecture Overview

The system contains five high-level boxes:

- `audio_input` Gets audio data from a microphone, and converts it into the required format for the next boxes.
- `mpeg_encoder` Encodes the audio stream into MPEG layer 1, for transmission.
- `communication_channel` A communication channel can be also modeled as a box, in a similar spirit to SystemC or SpecC. Note however that it can also be considered as a type converter, because it may be seen as an interface adaptor.
- `mpeg_decoder` Decodes the MPEG stream into raw audio data.
- `audio_output` Reconstructs the audio signal from the digital data.

Every box is copy-compatible with the next one.

Note that there is no sharing of resources. In real systems, it is also very common that copy operations are much more frequent than bind or select. This is good, because copy is the operation with less overhead.

B. Detailed Description

To show how complex boxes can be divided into simpler boxes, we are going to describe the `audio_input` box in detail. We expand this box because it contains analog and mixed-signal devices, and it is simple enough to be described as an example. For example, the MPEG encoder has eight internal boxes, and one of them is also divided into six more boxes.

The `audio_input` box, as any other box, has an input and an output. The input is not used, so it can be the destination of any copy operation (every box is copy-compatible with this one, because the input is discarded).

This box is divided into three smaller boxes: microphone, low-pass filter, and analog-to-digital converter.

The microphone is also modeled as a box that discards the input. The type of the microphone box output is a continuous analog signal. Therefore, it can only be copied to boxes with an analog input with similar physical attributes (two wire interface , matched input impedance) and compatible signal attributes (dynamic range, bandwidth). The only supported protocol for this box is a simple continuous data transfer.

The analog input filter is also a box whose input and output share similar attributes. The type of the input and output data is an analog signal with a physical interface consisting of two wires, and the only supported protocol is a continuous data transfer. Those properties make this box a suitable destination for copy operations from the microphone box. Therefore, data can be directly fed from the microphone output to the filter input.

The conditioned audio signal at the output of the analog filter must be converted into a digital bitstream using an A/D converter, which is another box. The input is again an analog signal, but in this case we are not interested in the variation of the signal, but in the value of the signal at the moment of the copy operation. Thus, we need a sample-and-hold to remember the value during the conversion process. The output is a 16-bit digital bus, with a control line to indicate when the output is valid.

We should also implement the copy operation for every link (micro-filter, filter-ADC, ADC-encoder).

C. Putting It All Together

When we have implemented all the boxes and the operations, we need to describe the behavior of the whole system. In our case, it would be basically a program like this one:

```
while (1) {
  copy (in, audio_input); // discarded
  copy (audio_input, mpeg_encoder);
  copy (mpeg_encoder,
        communication_channel);
  copy (communication_channel,
        mpeg_decoder);
  copy (mpeg_decoder, audio_output);
}
```

Of course, all those copy operations are implemented using the copy operations of the inner boxes. For example, copying from the `audio_input` box would be implemented with a program like this:

```
copy (audio_input, dest)
{
  while (1) {
    copy (in, micro); // discarded
    copy (micro, filter);
    copy (filter, adc);
    copy (adc, dest);
  }
}
```

D. HW-SW Cosynthesis

This paper concentrates on the hardware-software operating system and how it helps to reduce the overall complexity. To better understand how this fits into a complete cosynthesis system, see [8] and the FLECOS web page [4].

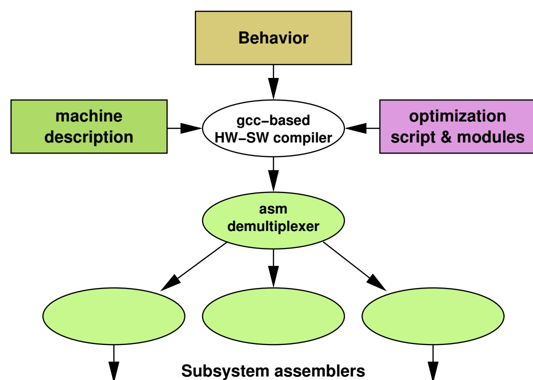

Fig. 2: Overview of the FLECOS design system

Basically, the FLECOS system uses a GCC-based hardware-software compiler to map the behavior specification into the available resources, taking into account the constrains and design criteria. Figure 2 depicts this process. The compiler generates assembler instructions and system calls for each of the subsystems. The low-level synthesis tools for each subsystem are responsible of mapping the system calls into hardware and/or software resources.

There are multiple implementation schemes to introduce a hardware-software operating system in a design flow, this is just one of them. Another possibility would be a library-based preprocessor phase as presented in [7].

IV. Related Work

Traditionally, the notion of operating systems have been associated to the management of dynamic resources. Some researchers coined the term *hardware operating system* [2], [6] to refer to partitioning methodologies focused on the exploitation of dynamic reconfiguration capabilities of some field-programmable devices.

However, the most important aspect of an operating system, as given by any definition, is that it abstracts and multiplexes the resources, and this abstraction simplifies the development of new applications. This is true even for hardware systems with only static and very restricted resources. To the best of our knowledge, these properties of the operating systems have never been used before to offer a unified and homogeneous view of a complete heterogenous system.

Our hardware-software operating system has some similarities with the SpecC methodology [5], and the multiple abstraction levels of SystemC [9]. However, our approach is not language-dependent, we make special emphasis in the definition of common interfaces for heterogeneous devices, and we use the same abstractions as a software operating system.

The idea of a very simple operating system to abstract the available resources comes from modern, lightweight microkernels, such as the exokernel [3], from MIT, and Off++ [1].

V. Conclusions

We have proposed a powerful yet simple interface, borrowed from the *Off++* distributed microkernel, as the foundation of a flexible way to introduce component-based methodologies into hardware-software codesign. We make special emphasis on the use of simple, homogeneous interfaces (the box interface) to hide the inherent complexity of current designs.

The main benefits of this approach are:

1) Faster and easier design of complex, distributed, hardware-software embedded systems, with simple and homogeneous interfaces.
2) More reuse opportunities and easier reuse, thanks to a common interface and the type conversion mechanism.
3) The overhead introduced by the higher abstraction level can be really low.

Thus, the design of complex embedded systems can be greatly improved with the use of hardware-software operating systems.

VI. Future Work

Off++ is also a distributed adaptable microkernel. It does not consider only the resources in a single node, it exports all the resources in the network. This gives multiple opportunities for resource sharing, fault tolerance, etc. We want to apply these ideas also for hardware-software systems.

We also want to implement common abstractions, such as multiple execution contexts, shared memory, and message passing, on top of the hardware-software box interface.

References

[1] Francisco J. Ballesteros, Fabio Kon, and Roy H. Campbell. A Detailed Description of Off++, a Distributed Adaptable Microkernel. Technical Report UIUCDCS-R-97-2035, University of Illinois at Urbana-Champaign, August 1997.

[2] G. Brebner. A virtual hardware operating system for the xilinx xc6200. In R. W. Hartenstein and M. Glesner, editors, *6th International Workshop on Field-Programmable Logic and Applications (FPL)*, Field-Programmable Logic: Smart Applications, New Paradigms and Compilers, pages 156–165, Berlin, September 1996. Springer-Verlag.

[3] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the 15th Symposium on Operating System Principles (SOSP)*, 1995.

[4] FLECOS web site. [on-line]. Available from WWW: `http://arco.inf-cr.uclm.es/flecos.html`.

[5] Daniel D. Gajski, Rainer Dömer, and Jianwen Zhu. IP-centric methodology and design with the SpecC language. Technical report, University of California, Irvine, 1999.

[6] P. Merino, J.C. López, and M. Jacome. A Hardware Operating System for Dynamic Reconfiguration of FPGAs. In *Proc. 8th International Workshop on Field Programmable Logic and Applications (FPL'98)*, August 1998.

[7] José M. Moya, Santiago Domínguez, Francisco Moya, and Juan Carlos López. A Flexible Specification Framework for Hardware-Software Codesign. In *Design Automation and Test in Europe*, 2000.

[8] José M. Moya, Francisco Moya, Santiago Domínguez, and Juan Carlos López. Multi-Language Specification of Heterogeneous Systems. In *Forum on Design Languages*, 2000.

[9] Open SystemC Initiative. [on-line]. Available from WWW: `http://www.systemc.org/`.

[10] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, 1987.