A Strategy for Real-Time Kernel Support in Application-Specific HW/SW Embedded Architectures

Steven Vercauteren Bill Lin Hugo De Man

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium
E-mail: {vercaut,billlin,deman}@imec.be
Tel: +32/16/28.15.25 ; Fax: +32/16/28.15.15

Abstract

Heterogeneous embedded multiprocessor architectures are becoming more prominent as a key design solution to today's microelectronics design problems. These application-specific architectures integrate multiple software programmable processors and dedicated hardware components together on to a single cost-efficient IC. In contrast to general-purpose computer systems, embedded systems are designed and optimized to provide specific functionality, using possibly a combination of different classes of processors (e.g. DSPs, microcontrollers) from different vendors. While these customized heterogeneous multiprocessor architectures offer designers new possibilities to tradeoff programmability, processing performance, power dissipation, and design turnaround time, there is currently a lack of tools to support the programming of these architectures. In this paper, we consider the problem of providing real-time kernel support for managing the concurrent software tasks that are distributed over a set of processors in an application-specific multiprocessor architecture. This is complementary to current research activities that aim to provide efficient retargetable code generation [8, 11, 10] for a broad range of embedded processors.

1 Introduction.

Telecommunication and multi-media computing are among the fastest growing segments of the microelectronics market today. These market sectors are being fueled by new emerging business and consumer applications that are now possible with recent advances in wireless communication, video-processing, and integrated networking technologies. The design of VLSI chips in these applications are often subject to stringent requirements in terms of processing performance and power dissipation. At the same time, programmability is becoming increasingly important for facilitating flexible designs that can be customized with differentiating features for use in multiple products. Further, strong economic pressures are demanding highly cost-efficient solutions that must be delivered in increasingly shorter time-to-market windows.

To facilitate flexible low-cost designs in short design time, emerging designs are based on heterogeneous embedded system architectures, that integrate software programmable components, e.g. DSP and microcontroller cores, together with dedicated hardware components on to a single cost-efficient IC. An example of a typical heterogeneous embedded processing system is depicted in Figure 1. The shaded boxes correspond to software programmable components and the non-shaded boxes correspond to hardware components. Programmability is introduced in these architectures (thus offering the desired flexibility in the design process), while maintaining most of the advantages of customized VLSI architectures (such as the potential to optimize the processing performance and power dissipation), thus forming a synergy between pure ASIC and programmable processor solutions.

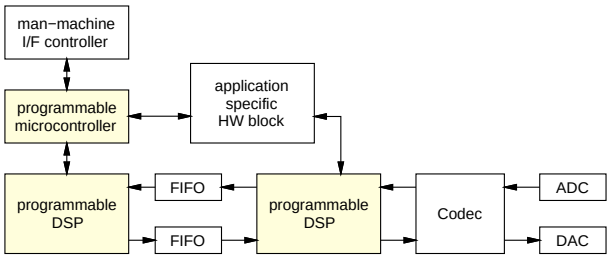

Fig. 1. Heterogeneous embedded system architectures.

While application-specific multiprocessor architectures offer designers new possibilities to tradeoff programmability, processing performance, power dissipation, and design turnaround time, there is currently a lack of tools to support the programming of these architectures. In this paper, we consider the problem of providing real-time kernel support for managing the concurrent software tasks that are distributed over these application-specific multiprocessor architectures. The problem is exacerbated by the fact that any solution to this problem must consider the inherent heterogeneity in these architectures; i.e., different classes of processors (e.g. DSP vs. microcontrollers) from different vendors may be combined with dedicated hardware components in order to optimize for specific functionality. This heterogeneity problem is partly addressed by current research efforts that aim to provide efficient retargetable code generation [8, 11, 10] for a broad range of embedded processors. These research activities complement our work here, which aims to offer a methodology for providing real-time kernel support in application-specific architectures.

While the design of real-time kernels is not new, earlier kernels were bulky and did not address well performance issues [4]. However, in recent years, a number of lightweight real-time kernels dedicated to embedded applications have emerged on the market [18, 19, 16, 17, 20, 15] that are smaller in size and are better tuned for performance (e.g. fast con-

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

text switch and interrupt service). They provide many high-level services to support the run-time coordination of concurrent software tasks that are distributed over a set of processors. Services include task scheduling and thread control, inter-process communication mechanisms like semaphores, mail-boxes, and queues, which significantly ease the implementation of distributed programs in software. Most kernel vendors aim to support a broad portfolio of processors (both DSPs and microcontrollers) by using a kernel architecture that isolates processor specific aspects so that minimal porting is required to support different processors.

However, the currently available real-time kernels are not without limitations. They typically assume a fixed pre-implemented multiprocessor architectures on printed circuit boards (e.g. VME-based boards [7]), They cannot directly be used for an application-specific architecture without manual porting of low-level services such as interprocessor communication, and they do not at all support mixed hardware/software solutions since functionalities for communicating and interacting with dedicated hardware components are entirely lacking. As a consequence, real-time kernels are rarely used for embedded system-on-silicon applications, despite the important functionalities that they provide. Instead, designers currently have to struggle with implementing manually low-level assembly routines to implement communication and task coordination functionalities that are often already a part of a real-time kernel. The implementation of these routines is a laborious and highly error proned task, without a clear methodology to structure the implementation process.

In this paper, we present a methodology for providing real-time kernel support for application-specific hardware/software embedded architectures. A key concept in our approach is the generation of a hardware/software communication abstraction layer to assist flexibility of programming. This is supported through a standard architecture, programming structure and communication protocol. The result of this work permits a reduced development cycle and can be used as guidelines for real-time kernel vendors to structure the architecture of their kernels and as guidelines for system CAD vendors to incorporate real-time kernel technology into their system design flow.

The remainder of this paper is organized as follows. In Sections 2 and 3, we describe our target implementation architecture model and the basic kernel architecture, respectively. In Section 4 we explain how real-time kernels can be ported to our target architecture. The possible communication scenarios are highlighted in Section 5. In Section 6 we discuss the implementation issues, and we demonstrate our approach by applying our methodology to an existing commercial real-time kernel family. Finally, conclusions are drawn in Section 7.

2 Target Implementation Architecture

In this section we describe our target implementation architecture model. In our model, the architecture is abstracted as an interconnection of *Processor Component Units* (PCUs) and

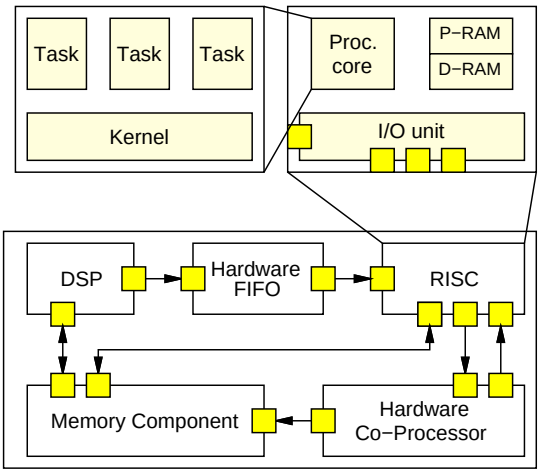

Fig. 2. Target Architecture Model.

point-to-point unidirectional or bidirectional channels, as depicted in Figure 2. A processor component unit can either be a hardware component or a software programmable component (e.g. DSP, ASIP, or micro-controller core). Our target architecture is therefore based on the following concepts: communication channels, hardware components and software components.

Communication Channels. Communication between the different component units is based on sending and receiving data to each other via communication channels. The channel communication semantics that we use is exactly that of Hoare's CSP rendezvous [6]. In this channel model, processor components can communicate with each other via explicit `send` and `receive` operations on a specified channel or via an intermediate shared memory component. We believe this model is sufficiently general because other communication models, such as buffered communication, can be mimicked by using intermediate components that implement that communication behavior. In Figure 2 this is illustrated by allocating a memory component that can be accessed by the DSP processor, the RISC processor and the hardware component `HW3`. This configuration will allow the three processor components to communicate with each other via shared memory. The same reasoning applies for the FIFO buffer that buffers the communication between the DSP processor and the RISC processor. The communication channels implement a common channel protocol at the circuit level that is consistent with the CSP rendezvous semantics. This channel protocol must be obeyed by all components by using explicit communication constructs. This ensures that all components can be integrated at the "implementation" level. However, the implementation of a channel may differ depending on the clocking relationship of the communicating components. Therefore, a channel can be seen as an abstraction of the physical communication link between two component units.

Hardware Components. A hardware component unit can either be a "pre-designed" library component, including parameterized communication components like buffers, or a hardware processor that has still to be synthesized. In both

cases the hardware component contain internal memory. In the former case the library component can be in the form of synthesizable VHDL source code or already at the circuit level. To represent parameterizable components, the parameterization features of VHDL can be used. The main requirement is that all external communication with the outside world must be implemented using the common channel communication protocol. In the latter case, the designer only needs to declare the number of communication channels required for the hardware processor component, their directions, and the data width that they must support. A customized VHDL package is then generated that implements the communication channels connected to the specific hardware component. The designer can then "program" the hardware component by writing a VHDL program that uses `send` and `receive` operations provided by the VHDL package for external communication. An important class of hardware components is the class of memory components. They are key to our architecture model as they will allow for shared memory communication, as indicated above. Depending on the specified channel layout of the memory component, the number of ports, the storage size, the word width and other characteristics of the selected memory, a hardware layer is generated around the memory that will implement the necessary synchronizing and decoding logic. A memory component therefore consists of a memory itself and a hardware layer for arbitrating the different accesses to the memory and for decoding supplied addresses, if necessary.

Software Components. In the case of a software programmable component, the processor component unit consists of the processor core itself, an internal memory structure for storing the program instructions and run-time data, and a hardware I/O unit that implements the communication interface to its external environment. The I/O unit acts as a "hardware wrapper" that effectively encapsulates a software programmable component into a hardware component. In fact, the hardware I/O unit implements the CSP rendezvous communication channels according to the common circuit protocol, for interconnection with the other components. In this approach, a software components is seen to the other components simply as another hardware processor component. The I/O unit itself is driven by the processor core via the software that executes on it. In this paper we assume the software running on the software component is managed by a real-time multi-tasking kernel. Based on the specified channel layout, the I/O unit and the inter-processor communication software routines of the kernel can be automatically generated, partly based on a parameterized library solution.

3 Basic Kernel Architecture

In this work we propose the use of multiple autonomous kernels with one kernel associated with each software processor component. In this case the kernel on each software processor component only manages the application tasks that are mapped to it. This is graphically shown in Figure 3. To have the same semantics across all kernels in the system, the same

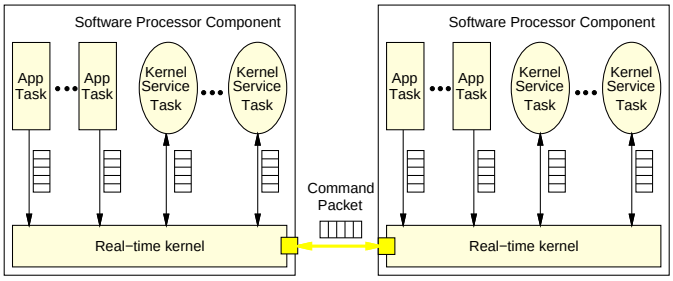

Fig. 3. Basic Kernel Architecture

kernel must be used for each software processor component. This is a viable approach as today there exist already commercially available real-time multitasking kernels that have been ported to a large number of popular RISC processors, DSP's, and micro-controllers [18, 19, 16, 17, 20, 15].

Firstly, the kernel is responsible for scheduling the different application tasks, handling communication between the application tasks, and synchronizing the application tasks with each other and with external events. To provide support for real-time behavior a kernel typically uses a *preemptive, priority driven scheduling* scheme for managing the timely execution of the different application tasks. In this scheme a priority is given to each application task. At any given instant the highest priority application task is granted execution control of the software processor by pre-emptying the application task that was currently executing. The thread of control associated with an application task is blocked if it has to wait for a synchronizing event from another application task, or from an source external to the processor.

Secondly, the kernel also provides a subroutine interface to a number of predefined kernel service tasks, that are called "kernel services". These kernel services provide, amongst others, resource protection, memory (de)allocation and communication and synchronization between application tasks. They must be allocated by the user and can be moved anywhere in the network of software processors without any changes to the application code. To support this topology transparency, a kernel specific communication protocol is maintained between the different kernels in the sense that all information handled by the kernels is packetized. In this scheme a task issues a kernel service by sending a command packet via the kernel to the relevant kernel task. This command packet contains information about the requested kernel service, a processor node identifier, argument data, etc. The different autonomous kernels then communicate by sending command packets to each other. Therefore, the different kernels also provide I/O communication functionalities and support for routing a command packet to its destination processor.

4 Parameterizing the Kernel

4.1 Kernel Architecture Template

In order to support the programmer with standard communication and synchronization primitives, a kernel typically has a layered architecture. We assume a kernel template that is divided up into two layers: a *scheduling layer* and a *communica-*

tion layer. In this architecture template, the scheduling layer is responsible for scheduling the different tasks that are mapped to it, while the actual I/O communication is done in the communication layer. This layer makes use of the processor specific I/O facilities and the communication bus characteristics. In fact, the communication layer abstracts away the architecture used and provide facilities for implementing the desired standard primitives somewhat akin to the conventional layered approach in computer communication and networking.

Real-time multi-tasking kernels, as they are used today, assume a fixed pre-implemented multiprocessor architecture. They cannot directly be used for a custom processor-bus architecture without manual porting the communication layer. For a custom system-on-silicon solution, manual porting of low level services and low level adaptations to the communication bus protocol are required. They currently do not at all support mixed hardware/software solutions since functionalities for communicating and interacting with hardware processors are entirely lacking.

By porting the different kernels to our target architecture, we standardize on the architecture and on the bus protocol used. As described in Section 2, our target architecture can be seen as interconnection of PCU's and point-to-point unidirectional or bidirectional channels, that are implemented according to a common channel protocol at the circuit level. Because of the standardized communication protocol, the design of the low level communication services is reduced to a one time effort, that can be saved away in a library. Using this library, we can configure the kernel communication layer as a parameterizable software template. For a specific software processor component, we consider the communication layer parameterized in terms of the number of channels that the software processor component is connected to, their directions and the data with that they must support. Once the user has defined the channel layout of the final architecture, the communication layer can then be customized. Some generic functions used in the predefined kernel tasks and the kernel itself are written in C and are portable across platforms via re-compilation. Other functions, especially in the communication layer, will be implemented in assembly for efficiency reasons. These functions are not entirely portable, as they must be rewritten for each processor. In our approach this low-level porting is reduced to a one time effort as all the processor specific functionalities are implemented and stored in a parameterized library, assuming our target architecture model. This strategy broadens the application domain of real-time kernels significantly, as their concurrent programming features can be used for the design of application-specific embedded hardware/software systems in a time efficient way.

4.2 Details of Communication Layer

In our kernel architecture, the communication layer will do the actual communication with the outside world, via communication channels that are connected to the other components. In constructing the communication layer, two types of channels

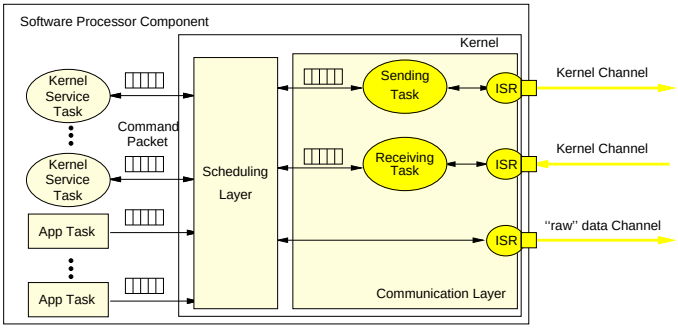

Fig. 4. Kernel architecture with Communication Layer expanded

can then be distinguished: *kernel channels* and *data channels*.

Kernel Channels. Kernel channels are exclusively used for communication and synchronization between the different kernels and cannot be used directly by the designer. A special communication task is assigned to each kernel channel, together with an Interrupt Service Routine (ISR). In Figure 4 a situation is depicted of a software processor component connected to two opposite directed kernel channels. In this example a sending task, a receiving task and two ISR's are assigned to the two kernel channels. These special communication tasks receive command packets from the scheduling layer and implement a state machine to decide what to do with a specific command packet. For kernel services that require argument parameters, the corresponding command packet will contain a memory pointer to the argument data. As in our model each component can have its own local memory, the communication tasks will then issue a complete interprocessor copy of the argument data, if necessary. In this scheme, the communication tasks only implement the kernel specific protocol by processing and interpreting the command packets in a correct way. The lowest level communication is handled at the "ISR layer". The implementation of the ISR's will indeed reflect the processor specific I/O facilities and the communication bus characteristics. These routines are responsible for marshaling and de-marshaling the command packets and possible argument data. Depending on the architecture of the hardware template surrounding the processor core, the ISR's will implement send and receive routines using the memory "read" and "write" instructions if memory-mapped I/O is used for the specific channel, or the corresponding special programmed I/O instructions if instruction-programmed I/O is used. Therefore different types of ISR's exist, that are saved away in a library.

Data Channels. Data channels are exclusively used by the application tasks for direct communication with a neighboring processor component. Some predefined kernel tasks provide services for sending and receiving data explicitly to and from a specified data channel. Since in this case the specified data channel will transfer "raw" data, no special communication tasks are required as data channels will never transport command packets. However, data transfer is still based on an interrupt-driven I/O scheme. Therefore an ISR is as-

signed to each data channel. In Figure 4 a situation is depicted of a software processor component connected to one outgoing data channel. From the ISR's perspective there is no difference with a kernel channel.

From this discussion, it is clear that we can configure the kernel communication layer as a software template, parameterized in terms of the number and type of ISR's, sending tasks and receiving tasks. This is due to our target implementation architecture model, as the standardized bus protocol will fix the possible ISR types. Once the designer has defined the final target architecture, we automatically customize the kernel communication layer.

5 Communication Scenarios

In this section we highlight the possible communication scenarios that can exist when providing real-time support in our target implementation architecture model. The following cases can be distinguished:

- **Case 1.** An application task calls a kernel service located on the same processor.
- **Case 2.** An application task calls a kernel service located on another processor.
- **Case 3.** An application task calls a kernel service issuing an explicit data transfer on a data channel connected to a software processor component.
- **Case 4.** An application task calls a kernel service issuing an explicit data transfer on a data channel connected to a hardware processor component.
- **Case 5.** Communication between two hardware components.

Case 1. In this case, an application task will request a kernel service from a kernel service task that is residing on the same software processor component. This kernel service is requested by calling a predefined subroutine that sends a command packet to the kernel that will provide the requested service and reschedule the application tasks afterwards. In this scheme all communication takes place at the software level.

Case 2. In this case a communication takes place between two software processor components. An application task calls a predefined subroutine requesting a kernel service from a kernel service task residing on another software processor component. The kernel inspects the command packet to verify whether the requested service is located on the host processor itself or on a other software processor component. If indeed the requested service is located elsewhere, the command packet is sent along a kernel channel to reach the software processor component that hosts the desired kernel task. This kernel task will then implement the requested service.

Case 3. An application task exchanges "raw" data with an application task on a neighboring software processor component in a explicit manner. This means an application task issues a kernel service specifying the data channel and the data that must be transferred along that channel. This communication is non-blocking in the sense that the application task can proceed immediately after issuing the communication. When the requested data transfer has completed an event is generated by the specific ISR that can be waited for by the application task. In this case no command packets are sent between the two software processors components.

Case 4. In this case a communication takes place between an application task residing on a software processor component and an "application task" located on a neighboring hardware processor component. From the software programmer's perspective there is no difference with Case 3. The hardware programmer on the other hand can program external communications on the specific application channel by issuing `send` and `receive` operation provided by a customized VHDL package, as already described in Section 2.

Case 5. In this case two hardware processor components are communicating with each other via a data channel. From the hardware programmer's perspective, there is no difference with Case 4.

6 Experimentation

The proposed techniques that have been presented in this paper are part of *Symphony*, a system integration tool-box within a larger heterogeneous system co-design environment called CoWare under construction at IMEC [3].

To evaluate our strategy we studied an existing commercially available real-time multi-tasking kernel, called Virtuoso, provided by Eonics Systems Inc. Currently Virtuoso provides support for developing real-time embedded software applications on PC and VME-based [7] multi-processor boards of different vendors. It assumes multiple autonomous kernels with one kernel associated with each processor. It provides many high-level services to support the run-time coordination of concurrent tasks that are distributed over a set of processors. Services include task scheduling and thread control, high-level interprocess communication mechanisms like semaphores, mailboxes, and queues, which significantly ease the implementation of distributed programs. However, it is currently targeted towards a limited set of "fixed pre-implemented" boards. For other commercially available boards or custom developed boards, custom porting is required, but this can take considerable time. Therefore, separate kernel versions are designed for each processor and for each board in which it used. This design practice prevented us currently from using the Virtuoso product family into our system-on-silicon design trajectory.

We applied the strategy presented in this paper for using the Virtuoso kernel into a custom embedded architecture, involving ARM RISC cores [14]. To incorporate the ARM-6 RISC processor core into an application-specific heterogeneous architecture, we constructed a parameterized hardware template architecture, consistent with our target implementation architecture model described in Section 2, as well as a parameterized kernel communication layer template that is graphically

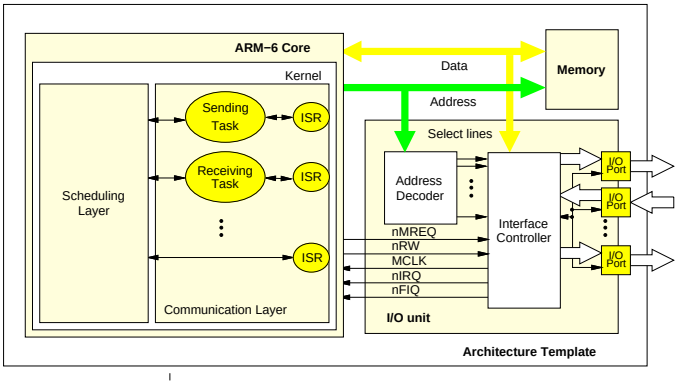

Fig. 5. Parameterized template for the ARM-6 RISC core.

depicted in Figure 5.

For the hardware template architecture we have chosen a memory mapped I/O scheme. The hardware template contains a address decoder and an interface controller that are parameterized in terms of the number of supported channels, their directions and data width. For this we use the parameterization features of VHDL. The same reasoning applies for the kernel communication layer, that is parameterized in terms of the number of ISR's, sending tasks and receiving tasks. When the user freezes the target implementation, our prototype tool not only customizes the hardware template architecture but also customizes the kernel communication layer, based on the specified channel configuration. Our parameterized architecture approach will allow to integrate the ARM-6 RISC processor core into application-specific hardware/software embedded architecture, while providing real-time kernel support for managing the execution of the different tasks running on the processor core(s) and for high-level interfacing with the hardware blocks.

7 Conclusion

In this paper we presented a strategy for providing real-time support in application-specific hardware/software architectures. Our approach is based on a simple scalable target architecture. We showed that the concurrent programming features of a real-time kernel can be leveraged to the design of systems-on-silicon, when ported to our architecture model. We also plan to expand our support for a larger portfolio of processor cores and to develop a better kernel customization strategy. We will also investigate how our real-time support strategy can be leveraged by emerging software synthesis approaches [1, 2, 5, 9, 13].

Acknowledgments

We would like to thank K. Van Rompaey, D. Verkest, I. Bolsens, G. Goossens, F. Catthoor, B. Gyselinckx, J. Silva, C. Ykman, T. Kolks, E. Verhulst for numerous insightful discussions on the system integration problem.

References

[1] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. A formal methodology for hardware/software codesign of embedded systems. *IEEE Micro*, August 1994.

[2] P. H. Chou, R. B. Ortega, G. Borriello. The Chinook Hardware/Software Co-Synthesis System, *International Symposium on System Synthesis*, September 1995.

[3] H. De Man, I. Bolsens, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest. Co-design of DSP systems. *NATO ASI Hardware/Software Co-Design*, Tremezzo, June 1995.

[4] K. Ghosh, et al. A Survey of Real-Time Operating Systems. *Technical Report nr. GIT-CC-93/18*, Georgia Institute of Technology, Atlanta, Georgia, February 1994.

[5] R. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems. *Computers and Electrical Engineering*, 10(3)29–41, September 1993.

[6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[7] IEEE, *IEEE/ANSI Standard 1014, Versatile Backplane Bus: VMEbus*, IEEE Service Center, Piscataway, NJ, 1987.

[8] D. Lanneer, J. Van Praet, K. Schoofs, W. Geurts, A. Kifli, F. Thoen, and G. Goossens. Chess: retargetable code generation for embedded processors. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Boston, 1995.

[9] R. Lauwereins et al. Grape-II: A system level prototyping environment for DSP applications. *IEEE Computer*, pages 35 – 43, February 1995.

[10] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang. Code optimization techniques for embedded DSP microprocessors. *ACM/IEEE DAC*, June 1995.

[11] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala. FlexWare : a flexible firmware development environment for embedded systems. In *Code generation for embedded processors*, pp. 67–84, Kluwer Acad. Publ., Boston, 1995.

[12] A. W. Roscoe, C. A. R. Hoare. Laws of Occam Programming. *Theoretical Computer Science*, 60, 177–229, 1988.

[13] F. Thoen, M. Cornero, G. Goossens, and H. De Man. Real-time multi-tasking in software synthesis for information processing systems. *ISSS'95*, September 1995.

[14] A. van Someren and C. Atack. *The ARM RISC Chip, A programmer's Guide*. Addison-Wesley Publishing Company, 1994. ISBN 0 201 40695 0.

[15] EOS - Etnoteam Operating System, Etnoteam S.p.A.

[16] pSOS+ - Integrated Systems.

[17] VxWorks - WindRiver Systems.

[18] J. F. Ready, "VRTX: A Real-Time Operating System for Embedded Microprocessor Applications," IEEE Micro, pp. 8-17, Aug., 1986.

[19] SPECTRON Microsystems, "SPOX - the DSP Operating System," 5266 Hollister Av., Santa Barbara CA 93111 (U.S.A), Jun., 1992.

[20] E. Verhulst, "Virtuoso: Providing Submicrosecond Context Switching on DSPs with a Dedicated Nano Kernel," International Conference on Signal Processing Applications and Technology, Santa Clara, Sept., 1993.