Systematic and Automated Multiprocessor System Design, Programming, and Implementation

Hristo Nikolov, *Student Member, IEEE*, Todor Stefanov, *Member, IEEE*, and Ed Deprettere, *Fellow, IEEE*

***Abstract*—For modern embedded systems in the realm of high-throughput multimedia, imaging, and signal processing, the complexity of embedded applications has reached a point where the performance requirements of these applications can no longer be supported by embedded system architectures based on a single processor. Thus, the emerging embedded system-on-chip platforms are increasingly becoming multiprocessor architectures. As a consequence, two major problems emerge, namely how to design and how to program such multiprocessor platforms in a systematic and automated way in order to reduce the design time and to satisfy the performance needs of applications executed on such platforms. As an efficient solution to these two problems, in this paper, we present the methodology and techniques implemented in a tool called Embedded System-level Platform synthesis and Application Mapping (ESPAM) for automated multiprocessor system design, programming, and implementation. ESPAM moves the design specification and programming from the Register Transfer Level and low-level C to a higher system level of abstraction. We explain how, starting from system-level platform, application, and mapping specifications, a multiprocessor platform is synthesized, programmed, and implemented in a systematic and automated way. The class of multiprocessor platforms we consider is introduced as well. To validate and evaluate our methodology, we used ESPAM to automatically generate and program several multiprocessor systems that execute three image processing applications, namely Sobel edge detection, Discrete Wavelet Transform, and Motion JPEG encoder. The performance of the systems that execute these applications is also presented in this paper.**

***Index Terms*—Automated programming of MultiProcessor Systems-on-Chip (MPSoC), design automation for MPSoC, system-level design.**

I. Introduction

MOORE'S law predicts the exponential growth over time of the number of transistors that can be integrated in a single chip. Not only must the intrinsic computational power of a chip be used efficiently and effectively, but also the time and effort to design a system containing both hardware and software must remain acceptable. Unfortunately, most of the current methodologies for multiprocessor system design are still based on register transfer level (RTL) platform descriptions created by hand using, for example, VHSIC Hardware Description Language (VHDL). Such methodologies were effective in the past when platforms based only on single-processor or processor–coprocessor architectures were considered. However, the applications and platforms used in many of today's new system designs are mainly based on heterogeneous Multi-Processor Systems-on-Chip (MPSoCs) architectures. As a consequence, the designs are so complex that traditional design practices are now inadequate, because creating RTL descriptions of complex MPSoCs is error prone and time consuming. The complexity of high-end computationally intensive applications in the realm of high-throughput multimedia, imaging, and digital signal processing exacerbates the difficulties associated with the traditional hand-coded RTL design. Moreover, to execute an application on MPSoC, the system has to be programmed, which implies writing software for each of the processors using languages such as C/C++. In recent years, a lot of attention has been paid to the building of MPSoCs. However, insufficient attention has been paid to the development of concepts, methodologies, and tools for the efficient programming of such systems so that the programming still remains a major difficulty and challenge [1]. Today, system designers experience significant difficulties in programming MPSoCs because the way an application is specified by an application developer, which is typically as a sequential program, does not match the way multiprocessor systems operate, i.e., multiprocessor systems require parallel application specifications.

Manuscript received January 24, 2007; revised June 1, 2007. This work was supported by the Dutch Technology Foundation (PROGRESS/STW) under Project Artemisia (LES.6389). This paper was recommended by Associate Editor S. A. Edwards.

The authors are with the Leiden Institute of Advanced Computer Science, Leiden University, 2333 Leiden, The Netherlands (e-mail: nikolov@liacs.nl; stefanov@liacs.nl; edd@liacs.nl).

Digital Object Identifier 10.1109/TCAD.2007.911337

A. Problem Description

For all the reasons stated above, we conclude the following.

1) The use of an RTL system specification as a starting point for multiprocessor system design methodologies is a bottleneck. Although the RTL system specification has the advantage that the state-of-the-art synthesis tools can use it as an input to automatically implement a system, we believe that a system should be specified at a higher level of abstraction called *system level*. This is the only way to solve the problems caused by the low-level RTL specification. However, moving up from the detailed RTL specification to a more abstract system-level specification opens a gap that we call *Implementation Gap*. Indeed, on one hand, the RTL system specification is very detailed and close to an implementation, which thereby allows an automated system synthesis path from RTL specification to implementation. This is obvious if we consider the current commercial synthesis tools where the RTL-to-netlist synthesis is very well developed and efficient. On the other hand, the complexity of today's systems forces us to move to higher levels of abstraction when designing

a system. However, we currently do not have mature methodologies, techniques, and tools to move down from the high-level system specification to an implementation. Therefore, the *Implementation Gap* has to be closed by devising a systematic and automated way to effectively and efficiently convert a system-level specification to an RTL specification.

2) Programming multiprocessor systems is a tedious, error-prone, and time-consuming process. On one hand, the applications are typically specified by the application developers as sequential programs using imperative programming languages such as C/C++ or Matlab. Specifying an application as a sequential program is relatively easy and convenient for application developers. However, the sequential nature of such specification does not reveal the available concurrency in an application because only a single thread of control is considered. In addition, the memory is global, and all data reside in the same memory source. On the other hand, system designers need parallel application specifications, because when an application is specified using a parallel model of computation (MoC), the programming of multiprocessor systems could be done in a systematic and automated way. This is so because the multiprocessor platforms contain processing components that run in parallel, and a parallel MoC represents an application as a composition of concurrent tasks with a well-defined mechanism for intertask communication and synchronization.

The facts discussed above suggest that to program an MPSoC, the system designers have to partition an application into concurrent tasks starting from a sequential program (delivered by application developers) as a reference specification. Then, they have to assign the tasks to processors and to write a specific program code for each processor. This activity consumes a lot of time and effort because the system designers have to study the application in order to identify the possible task-level parallelism that is available, and to reveal it. Moreover, an explicit synchronization for data communication between tasks is needed. This information is not available in the sequential program and has to be explicitly specified by the designers. Therefore, an approach and a tool support are needed for application partitioning and code generation to allow the systematic and automated programming of MPSoCs.

B. Paper Contributions

In this paper, we present our tool, i.e., Embedded System-level Platform synthesis and Application Mapping (ESPAM), and a design flow around it that implements our methods and techniques for systematic and automated multiprocessor platform design, programming, and implementation. These methods and techniques bridge the gap between the *system-level* specification and the RTL specification in a particular way that we consider as the main contribution of this paper. More specifically, with ESPAM, a system designer can specify a multiprocessor system at a high level of abstraction in a short amount of time, for example, a few minutes. Then, ESPAM refines this specification to a real implementation, i.e., generates a synthesizable (RTL) hardware (HW) description of the system and software (SW) code for each processor in a systematic and automated way, thereby closing in a particular way the implementation gap mentioned earlier. This reduces the design and programming time from months to hours. As a consequence, an accurate exploration of the performance of alternative multiprocessor platforms becomes feasible at implementation level in a few hours.

Our methods and techniques to closing the implementation gap are based on the underlying programming model and system-level platform model we use. ESPAM targets data-flow-dominated (streaming) applications for which we use the Kahn Process Network (KPN) [2] MoC as a programming (application) model. Many researchers [3]–[8] have already indicated that the KPNs are suitable for the efficient mapping onto multiprocessor platforms. In addition to that, by carefully exploiting and efficiently implementing the simple communication and synchronization features of a KPN, we have identified and developed a set of generic parameterized components that we call a platform model. We consider our platform model an important contribution of this paper because our set of components allows system designers to quickly and easily specify (construct) many alternative multiprocessor platforms that are systematically and automatically implemented and programmed by our tool ESPAM. The automated programming of our multiprocessor platforms is enabled by using a parallel programming model, namely KPN MoC. However, decomposing an application into a set of concurrent tasks is one of the most time-consuming jobs imaginable [1]. Fortunately, our programming approach integrates a tool that we have developed for the automatic derivation of KPN specifications from applications specified as sequential C programs [9].

C. Scope of Work

In this section, we outline the assumptions and restrictions regarding our work presented in this paper. Most of them are discussed in detail, where appropriate, throughout this paper.

1) Applications: One of the main assumptions for our work is that we only consider data-flow-dominated applications in the realm of multimedia, imaging, and signal processing, which naturally contain tasks communicating via streams of data. Such applications are very well modeled by using the parallel data flow MoC called KPN. The KPN model we use is a network of concurrent autonomous processes that communicate data in a point-to-point (P2P) fashion over bounded first-in–first-out (FIFO) channels using a blocking read/write on an empty/full FIFO as the synchronization mechanism.

2) Multiprocessor Platforms: We consider multiprocessor platforms in which only programmable processors are used as processing components, and they communicate data only through distributed memory units. Each memory unit can be organized as one or several FIFOs. The data communication and synchronization among processors are realized by blocking read and write SW primitives. Such platforms very well match and support the KPN operational semantics, thereby achieving high performance when the KPNs are executed on the platforms. Moreover, supporting the operational semantics of a

KPN in our platforms, i.e., the blocking mechanism, allows the processors to be self-scheduled. It means that there is no need for a global scheduler component in our platforms. The processors in our platforms can be connected either by a crossbar switch (CBS), or a P2P network, or a shared bus (ShB). If the number of processors in a platform is less than the number of processes of a KPN, then some of the processors execute more than one process. For each of those processors, we do not use a multithreading operating system (OS) to execute the processes mapped onto it in order to avoid execution overheads due to context switching. Instead, our tools schedule these processes at compile time and generate a program code for a given processor which code does not require/utilize an OS.

3) Tool Inputs: Our ESPAM tool accepts three specifications as input, i.e., platform specification, application specification, and mapping specification. The platform specification is restricted in the sense that it must specify a platform that consists of components taken from a predefined set of components. This set ensures that many alternative multiprocessor platforms can be constructed, and all of them fall into the class of platforms we consider (see above). The mapping specification can specify one-to-one and/or many-to-one mappings of processes onto processors. Based on this mapping, ESPAM automatically determines the most efficient mapping of FIFO channels onto distributed memory units. The application specification is a KPN derived from an application written as a static affine nested loop program (SANLP) in C. SANLP is a sequential program with some restrictions, see Section IV-A. The restrictions allow us to develop tools for the automated derivation of KPNs from SANLPs, as described in Section IV. The automated partitioning of a SANLP into processes can be done only at function boundaries, i.e., the programmer divides the SANLP into functions, thus guiding the granularity of the automatically derived processes. Many applications in the domain we consider (see above) can be represented as SANLPs.

D. Related Work

The Compaan/Laura design flow presented in [3] is very similar to the work described in this paper. It uses KPNs as an application model for the automated mapping of applications that target field-programmable gate array (FPGA) implementations. A KPN specification is automatically derived from a sequential program written in Matlab [10], [11] and implemented as a network of dedicated HW IP cores on an FPGA [12]. A major difference with our approach is that systems containing programmable processors are not considered in [3] and [10]–[12]. Although the KPN model we use is the same as in [10] and [11], we have developed different and improved techniques for the automated derivation of KPNs [9]. First and importantly, our techniques allow for the automated computation of efficient buffer sizes of the communication FIFO channels that guarantee deadlock-free execution of our KPNs. In contrast, the computation of the buffer sizes of FIFOs is not considered at all in the work presented in [10] and [11]. Second, we start from a fully functional application specification written as a SANLP in C, which allows our tools to automatically generate fully functional KPNs in C or C++ and to program fully automatically our multiprocessor platforms. In contrast, with the tools presented in [10] and [11], a fully automated programming is not considered.

The Eclipse work [13] defines a scalable architecture template for designing stream-oriented MPSoCs using KPN MoC to specify and map data-dependent applications. The Eclipse template is slightly more general than the templates presented in this paper. However, the Eclipse work lacks an automated design and implementation flow. In contrast, our work provides such automation starting from a high-level system specification.

The system-level semantics for system design formalization is presented in [14]. It enables design automation for synthesis and verification to achieve a required design productivity gain. Using Specification, Multiprocessing, and Architecture models, a translation from behavior to structural descriptions is possible at a system level of abstraction. Our approach is similar, but in addition, it defines and uses application and platform models that allow an automated translation from the system level to the RTL level of abstraction.

In our automated design flow for MPSoC programming and implementation, we use a parallel MoC to represent an application and to map it onto alternative MPSoC architectures. A similar approach is presented in [15]. Jerraya *et al.* propose a design flow concept that uses a high-level parallel programming model to abstract hardware/software interfaces in the case of heterogeneous MPSoC design. The details are presented in [16] and [17]. In [16], a design flow for the generation of application-specific multiprocessor architectures is presented. This paper is similar to our approach in the sense that we also generate multiprocessor systems based on the instantiation of generic parameterized architecture components as well as communication controllers (CCs) to connect processors to communication networks. However, many steps of the design flow in [16] are manually performed. As a consequence, a full implementation of a system comprising four processors connected P2P takes around 33 h. In contrast, our design flow is fully automated, and a full implementation of a system comprising eight processors connected P2P, or via a crossbar (CB) or ShB, takes around 2 h.

In [17], Gauthier *et al.* present a method for the programming of MPSoCs by the automatic generation of application-specific OS and the automatic targeting of the application code to the generated OS. In the proposed method, the OS is generated from an OS library and includes only the OS services specific to the application. The input to the code generation flow consists of structural information about the MPSoC, allocation information (memory map of the MPSoC), and high-level task descriptions. By contrast, in our programming approach, we do not use OSs. For each processor of an MPSoC, our tool generates a sequential code that contains control (for communication, synchronization, and task scheduling) and application-specific code. Another major difference is that in our approach the allocation information (the memory map of a MPSoC) and the task descriptions are automatically generated.

The Multiflex system presented in [18] is an application-to-platform mapping tool. It targets multimedia and networking applications, and integrates a system-level design exploration framework. The relation to our work is that ESPAM also targets the mapping of multimedia and data streaming applications onto a particular MPSoC platform. A design space exploration is included in our design flow as well. However, in our design flow, we use KPNs as the parallel programming model instead of the symmetrical multiprocessing (SMP) and distributed system object component (DSOC) used in Multiflex. The benefit of using KPNs is related to the KPN model properties that allow us to derive KPNs in an automated way from applications specified as sequential programs. Multiflex does not support at all the automatic derivation of SMP or DSOC. In [18], a design time of two man-months is reported for an MPEG4 multiprocessor system. The design time includes manual application partitioning, automated architecture exploration, and optimization. In this paper, we show that by using our design flow, a complete design including partitioning, exploration, implementation, and programming of a similar multiprocessor system is achieved within 2 h.

A method for the automatic generation of embedded software is presented in [19]. The proposed design flow consists of several software refinement steps and intermediate models to generate an efficient ANSI C code from system specification written in a system level design language (SLDL). This paper is similar to our work in the sense that our ESPAM tool generates an efficient C/C++ code for processors in MPSoC. The difference is that some of the software refinement steps in [19] have to be manually performed, whereas our software generation is fully automated. Moreover, we generate software for processors in MPSoC starting from an application specified as a sequential program in the widely accepted C language. In [19], a designer has to specify an application using the specific SLDL language.

The Task Transaction Level (TTL) interface presented in [20] is a design technology for the programming of embedded multiprocessor systems. Our programming approach is similar to TTL in the sense that we also target streaming applications and we also use communication primitives. TTL is more flexible in the sense that it supports many communication primitives, but programming MPSoC by using TTL requires a lot of manual work that is hard (in some cases even impossible) to automate. In comparison, our programming model supports only two basic primitives, i.e., blocking read and blocking write, both in order. We do not see this as a limitation compared to TTL because these two primitives are sufficient to realize communication and synchronization in any streaming application. Moreover, supporting only two basic primitives allows us to fully automate the programming of MPSoCs as we will show in this paper.

Companies such as Xilinx and Altera provide approaches and design tools that attempt to facilitate the efficient implementations of processor-based systems on FPGAs. These tools are the Embedded Development Kit [21] for Xilinx chips and the System On a Programmable Chip builder [22] for Altera chips. A recent survey of multiprocessor solutions [23] shows that these state-of-the-art tools support only processor–coprocessor systems and shared memory bus-based multiprocessor systems, which cannot always meet the performance requirements of today's (streaming) applications. In contrast, this paper proposes a platform model that supports different communication topologies (not only ShB) and allows different types of processors to be connected in heterogeneous multiprocessor platforms. In addition, we use a parallel MoC to represent an application and to map it onto multiprocessor platforms. Exploiting the properties of our platform and application models allows for automated MPSoC synthesis and implementation, application-dependent self-scheduling of the platform resources, and fully automated MPSoC programming.

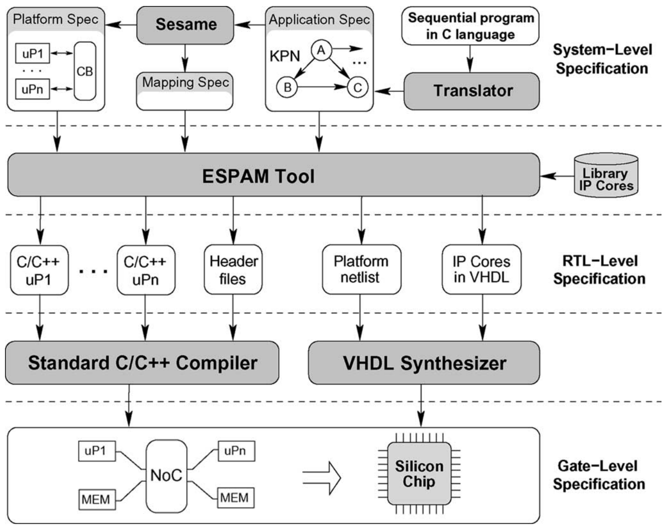


Fig. 1. System design flow.

II. DESIGN FLOW: OVERVIEW

In this section, we give an overview of our system design methodology, which is centered around the ESPAM tool that we have developed to close the implementation gap and the programming problem described in Section I-A. This is followed by a description of our system-level platform model and platform synthesis in Section III. In Section IV, we present our techniques for the automated derivation of KPNs and for computing the FIFO sizes. Then, in Section V, we discuss the automated programming of multiprocessor platforms, and in Section VI, we present the results that we have obtained using ESPAM. Section VII concludes this paper.

Our system design methodology is depicted as a design flow in Fig. 1. There are specifications at three different levels of abstraction in the flow, namely SYSTEM LEVEL, RTL LEVEL, and GATE LEVEL. The SYSTEM-LEVEL specification consists of the following three parts written in eXtensible Markup Language (XML) format.

1) *Application Specification* describes an application as a KPN, i.e., a network of concurrent processes communicating via FIFO channels. For applications initially specified as parameterized SANLPs in C, KPN descriptions can be automatically derived by using our translator tool [9].
2) *Platform Description* describes the topology of a multiprocessor platform. The platforms we consider are presented in Section III. Specifying a multiprocessor platform is a simple task that can be performed in a few minutes, because ESPAM requires a high-level platform

specification that does not contain any details about the physical interfaces of the components.

3) *Mapping Specification* describes the relation between all processes in the *Application Specification* and all components in the *Platform Specification*. In our case, describing the mapping in XML format is even simpler than describing a multiprocessor platform.

The platform and the mapping specifications can be written by hand or automatically generated after performing a design space exploration. For this purpose, we use the SESAME tool [5]. The input of SESAME is the KPN application specification. The output is a platform specification and a mapping specification that together represent the best mapping of an application onto a particular multiprocessor platform in terms of performance.

The SYSTEM-LEVEL specification is given as input to ESPAM. First, ESPAM constructs a platform instance from the platform specification and runs a consistency check on that instance. The platform instance is an abstract model of a multiprocessor platform because, at this stage, no information about the target physical platform is taken into account. The model only defines the key system components of the platform and their attributes. Second, ESPAM refines the abstract platform model to an elaborate (detailed) parameterized RTL model that is ready for an implementation on a target physical platform. We call this refinement process platform synthesis. The refined system components are instantiated by setting their parameters based on the target physical platform features. Finally, ESPAM generates a program code for each processor in the multiprocessor platform in accordance with the application and mapping specifications.

As output, ESPAM delivers a hardware (synthesizable VHDL code) description of the MPSoC and software (C/C++) code to program each processor in the MPSoC. The hardware generated by ESPAM, namely an RTL-LEVEL specification of a multiprocessor system, is a model that can adequately abstract and exploit the key features of a target physical platform at the RTL of abstraction. It consists of two parts: 1) *platform topology*, which is a netlist description that defines in greater detail the multiprocessor platform topology; and 2) *hardware descriptions of IP cores*, which contains the predefined and custom IP cores used in *platform topology*. ESPAM selects the predefined IP cores (processors, memories, etc.) from the *Library IP Cores* (see Fig. 1). Moreover, it generates the custom IP cores that are needed as glue/interface logic between components in the platform.

ESPAM converts the XML application specification to an efficient C/C++ code, including a code for implementing the functional behavior together with a code for the synchronization of the communication between processors. This synchronization code contains a memory map of the MPSoC and read/write synchronization primitives. They are inserted by ESPAM in the places of the processors' code where read/write access to a FIFO is performed. The program C/C++ code generated by ESPAM for each processor in the MPSoC is given to a standard GNU compiler collection (GCC) compiler to generate the executable code. With the hardware descriptions generated by ESPAM, a commercial synthesizer can convert the RTL-LEVEL specification to a GATE-LEVEL specification, which thereby generates the target platform gate level netlist (see the bottom part of Fig. 1). This GATE-LEVEL specification is actually the system implementation. The current version of ESPAM facilitates the automated multiprocessor platform synthesis and programming that targets the Xilinx FPGA technology, and thus, we use the development tools (a GCC compiler and a VHDL synthesizer) provided by Xilinx [21] to generate the final bit stream file that configures a specific FPGA. We use the FPGA platform technology only for prototyping purposes. Our ESPAM tool is flexible enough to target other physical platform technologies.

III. PLATFORM MODEL AND SYNTHESIS

In our design methodology, the platform model is a library of generic parameterized components. To support the systematic and automated synthesis of multiprocessor platforms, we have carefully identified and developed a set of computation and communication components. In this section, we give a detailed description of our approach to build a multiprocessor platform. The platform model contains the following.

1) *Processing Components*: Currently, our platform model supports only one type of processing component, i.e., a programmable processor. It has several parameters, such as *type*, *number of I/O ports*, *speed*, etc.
2) *Memory Components*: Memory components are used to specify the processors' local program and data memories, and to specify the data communication storages (buffers) between processors. Further, we will call the data communication storages *Communication Memories* (CMs). The important memory component parameters are *type*, *size*, and *number of I/O ports*. *Memory Controllers* (MCs) are used to connect the local program and the data memories to the processors.
3) *Communication Components*: We have developed a P2P network, a CBS, and a ShB component with several arbitration schemes. These *Communication* components are mutually exclusive and determine the communication network topology of a multiprocessor platform.
4) *Communication Controllers (CCs) and Links*: CCs are used as glue logic that realizes the synchronization between the processors at hardware level. A CC implements an interface between processing, memory, and communication components. Links are used to connect any two components in our system-level platform model.

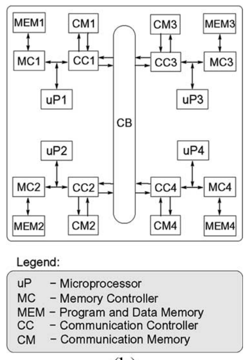
Using the components described above, a system designer can construct many alternative platforms by simply connecting processing, memory, and communication components. We have developed a general approach to connect and synchronize programmable processors of arbitrary types via a communication component. Our approach is explained below using an example of a multiprocessor platform. The system-level specification of the platform is depicted in Fig. 2(a). This specification, which is written in XML format, consists of three parts that define the processing components (four processors, lines 2–5), communication component (CB, lines 7–12), and links

```
1  <platform  name ="myPlatform">
     <processor name = "uP1"> <port  name = "IO1"/> </processor>
     <processor name = "uP2"> <port  name = "IO1"/> </processor>
     <processor name = "uP3"> <port  name = "IO1"/> </processor>
5    <processor name = "uP4"> <port  name = "IO1"/> </processor>

     <network name = "CB"  type = "Crossbar">
        <port  name = "IO1"/>
        <port  name = "IO2"/>
10      <port  name = "IO3"/>
        <port  name = "IO4"/>
     </network>

     <link name = "BUS1"/>
15      <resource  name = "CB"  <port  name = "IO1"/>
        <resource  name = "uP1"<port  name = "IO1"/>
     </link>
     <link  name = "BUS2"/>
        <resource  name = "CB"  <port  name = "IO2"/>
20      <resource  name = "uP2"<port  name = "IO1"/>
     </link>
     <link  name = "BUS3"/>
        <resource  name = "CB"  <port  name = "IO3"/>
        <resource  name = "uP3"<port  name = "IO1"/>
25   </link>
     <link  name = "BUS4"/>
        <resource  name = "CB"  <port  name = "IO4"/>
        <resource  name = "uP4"<port  name = "IO1"/>
     </link>
30 </platform>
```

(a)



(b)

Fig. 2. Example of a multiprocessor platform. (a) Platform specification. (b) Elaborate platform.

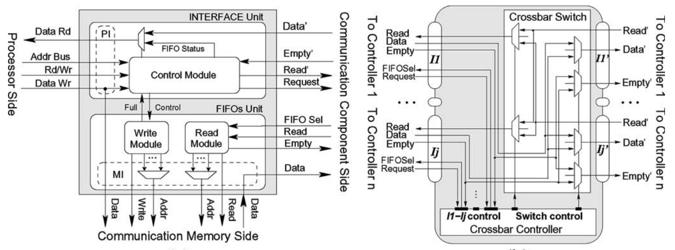


Fig. 3. Structure of the (a) CC and (b) CB components.

(lines 14–29). The links specify the connections of the processors to the communication component.

Notice that in the specification, a designer does not have to take care of the memory structures, interface controllers, and communication and synchronization protocols. Our ESPAM tool takes care of this in the platform synthesis as follows: First, the tool instantiates the processing and communication components. Second, it automatically attaches memories and MCs to each processor. Third, based on the type of the processors instantiated in the first step, the tool automatically synthesizes, instantiates, and connects all the necessary CCs and CMs to allow the efficient and safe (lossless) data communication and synchronization between components. The latter is guaranteed by the fact that the operational semantics of the KPNs, i.e., read–execute–write using the blocking read/write synchronization mechanism, is captured in and translated to the execution model of the multiprocessor platforms we propose. In our approach, the processors transfer data between each other through the CMs. Each processor writes only to its local CM and uses the communication component only to read data from all the other CMs. A processor blocks on writing if there is no room in its local CM, and a processor blocks when reading the other processors' CMs if data is not available or the communication resource is currently not available.

The elaborate platform generated by ESPAM is shown in Fig. 2(b). A CC connects a CM to the data bus of the processor (uP) it belongs to and to a communication component (CB). Each CC implements the processor's local bus-based access protocol to the CM for write operations and the access to the communication component for read operations. Each CM is organized as one or more FIFO buffers. The interprocessor synchronization in the platform is implemented in a simple and efficient way by blocking the read/write operations on empty/full FIFO buffers located in the CM.

KPNs assume unbounded communication buffers. Writing is always possible, and thus, a process blocks only on reading from an empty FIFO. In the physical implementation, however, the communication buffers have bounded sizes, and therefore, a blocking write synchronization mechanism is used as well. The problem of deciding whether a general KPN can be scheduled with a bounded memory is undecidable [24], [25]. However, in our case, this is possible because our process networks are derived from SANLPs that can be executed using a finite amount of memory. The scheduling of process networks using a bounded memory has already been discussed in [24] and [26]. Moreover, a number of tools and libraries have been developed for executing KPNs [27], [28]. In contrast to these approaches, our platform model does not require scheduling and run-time deadlock detection and resolution. The processing components in our platforms are self-scheduled following the KPN operational semantics, i.e., the KPNs are self-scheduled when executed on our platforms. In addition, we compute the buffer sizes of the FIFO channels (see Section IV-B) such that a deadlock-free execution of the KPNs on our platforms is guaranteed.

A. Processing Components

In our approach, we do not consider the design of processing components. Instead, we use the IP cores developed by third parties. Currently, for fast prototyping in order to validate our approach, we use the Xilinx VirtexII-Pro FPGA technology. Therefore, our library of processing components includes the two programmable processors supported by Xilinx. These are the *MicroBlaze* (MB) softcore processor and the *PowerPC* (PPC) hardcore processor.

B. CM Components

We implement the CMs of a processor by using dual-port memories. A CM is organized as one or more FIFO buffers. A FIFO buffer in a CM is seen by a processor as two memory locations in its address space. A processor uses the first location to read/write data from/to the FIFO buffer, thereby realizing interprocessor data transfer. The second location is used to read the status of the FIFO. The status indicates whether a FIFO is full (data cannot be written) or empty (data is not available). This information is used for interprocessor synchronization. The multi-FIFO behavior of a CM is implemented by the CC described below. However, if the CM contains only one FIFO, we use a dedicated FIFO component that simplifies the structure of the CC.

C. CC Component

The structure of the CC is shown in Fig. 3(a). It consists of two main parts, i.e., INTERFACE Unit and FIFO Unit.

The INTERFACE Unit contains: 1) the Control Module, i.e., an address decoder, FIFOs' control logic, and logic to generate read requests to the communication component and 2) the processor interface (PI) module that realizes the data bus protocol of a particular processor. The FIFO Unit implements the multi-FIFO behavior. The FIFO Unit also includes the memory interface (MI) module that realizes access to the CM connected to the CC [bottom part of Fig. 3(a)].

Recall that a processor can access the FIFOs located in the other processors' CMs via a communication component for read operations only. First, the processor checks whether there is any data in the FIFO that the processor wants to read from. When a processor checks for data, the INTERFACE Unit sends a request to the communication component for granting a connection to the CM in which the FIFO is located. A connection is only granted when a communication line is available and there is data in the FIFO. When a connection is not granted, the processor blocks until a connection is granted. When a connection is granted, the CC connects the data bus of the communication component [the upper part of the *communication component side* in Fig. 3(a)] to the data bus of the processor, and the processor reads the data from the CM where the FIFO is located. After the data is read, the connection has to be released. This allows other processors to access the same CM.

D. CB Communication Component

Our general approach to connect processors that communicate data through CM with FIFO organization allows the CB structure, which is shown in Fig. 3(b), to be very simple. It consists of two main parts, i.e., the crossbar switch (CBS) and the CB controller. The CBS implements unidirectional connections between CMs and processors—recall that a processor uses a communication component only to read data. Each processor and its local CM are connected to a CC, as shown in Fig. 2. The CC is connected to the CB using one I and one I' interface, as depicted in Fig. 3(b). Due to the unidirectional communications and the FIFO organization of CMs, the number of signals and busses that have to be switched by our CB is reduced. Since the addresses for accessing CMs are locally generated by the CCs, address busses are not switched through the CB. The CB switches only 32-bit data buses in one direction and two control signals per bus. These control signals are the Read strobe and the Empty status flag for a FIFO.

E. P2P Network

In P2P networks, the topology of the platform (the number of processors and the number of direct connections between the processors) is the same as the topology of the process network. Since there is no communication component such as a CB or a bus, there are no requests for granting connections, and there is no sharing of communication resources. Therefore, no additional communication delay is introduced in the platform. Because of this, the highest possible communication performance can be achieved in such multiprocessor platforms. Under the conditions that each CM contains only one channel and each processor writes data only to its local CM (in compliance with our concept), our ESPAM tool synthesizes a P2P network in the following (automated) way: First, for each process in the KPN, ESPAM instantiates a processor, together with a CC. Then, ESPAM finds all the channels that the process writes to. For each such channel, the tool instantiates a CM and assigns the channel to this CM. Finally, ESPAM connects the CM to the already instantiated processor. In Fig. 4(b), we give an example of a P2P multiprocessor platform generated by ESPAM. The multiprocessor platform implements the KPN depicted in Fig. 4(a), where each process is executed on a separate processor.

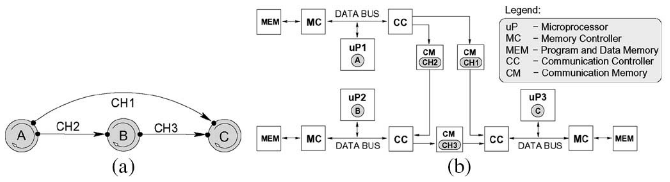


Fig. 4. P2P platform synthesis example. (a) Process network. (b) P2P architecture.

IV. AUTOMATED DERIVATION OF PROCESS NETWORKS

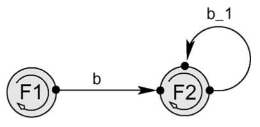
The KPN MoC has been widely studied by our group for more than seven years. The study resulted in techniques and tools [3], [10], [11] for the automated translation of SANLPs to KPN specifications. Although these techniques are very advanced, they generate KPNs with too many FIFO channels, which may lead to inefficient implementations. Moreover, they do not address at all the problem of what the FIFO buffer sizes should be. This is important because if the FIFO buffers are undersized, this may lead to a deadlock in the KPN behavior. Therefore, inspired by our previous work, we have recently developed techniques, which are implemented in a translator tool [9], for the *improved* derivation of KPNs. The input to the translator tool is a SANLP written in C, and the output is a KPN specification in XML format (see Fig. 1). In this section, we describe the underlying techniques of our translator tool. Below, in Section IV-A, we introduce the SANLPs and their restrictions, and explain how a KPN is derived based on a modified data flow analysis. We have modified the standard data flow analysis to derive KPNs that have less interprocess communication FIFO channels compared to the KPNs derived by using our previous work [10], [11]. Then, in Section IV-B, we show the techniques to compute the sizes of FIFO channels that guarantee deadlock-free execution of our KPNs mapped onto the multiprocessor platforms.

A. Modified Data Flow Analysis

SANLPs are programs that can be represented in the well-known polytope model [29]. That is, a SANLP consists of a set of statements and function calls, each possibly enclosed in loops and/or guarded by conditions. The loops need not be perfectly nested. All the lower and upper bounds of the loops as well as all the expressions in conditions and array accesses have to be affine functions of the enclosing loop iterators and parameters. The parameters are symbolic constants, i.e., their values

```
for ( int i=0; i<N; i++ )
    b[i] = F1( a[i], a[i+1] );
for ( int i=0; i<N; i++ ) {
    if  ( i>0 )  tmp = b[i-1];
    else         tmp = b[i];
    c[i] = F2( b[i], tmp );
}
```

(a)



(b)

Fig. 5. SANLP fragment and its corresponding KPN. (a) Example of a SANLP. (b) Corresponding KPN.

may not change during the execution of the program. The above restrictions allow a compact mathematical representation of SANLP through sets and relations of integer vectors defined by linear (in)equalities, existential quantification, and the union operation. In particular, the set of iterator vectors for which a function call is executed is an integer set called the *iteration domain*. The linear inequalities of this set correspond to the lower and upper bounds of the loops enclosing the function call. For example, the iteration domain of function F1 in Fig. 5(a) is $\{i | 0 \leq i \leq N-1\}$. The iteration domains form the basis of the description of the processes in our KPN model, as each process corresponds to a particular function call. For example, there are two function calls in the program fragment in Fig. 5(a), that represent two application tasks, namely F1 and F2. Therefore, there are two processes in the corresponding process network, as shown in Fig. 5(b). The granularity of F1 and F2 determines the granularity of the corresponding processes. The FIFO channels are determined by the array (or scalar) accesses in the corresponding function call. All accesses that appear on the left-hand side or in an address-of (&) expression for an argument of a function call are considered to be *write accesses*. All other accesses are considered to be *read accesses*.

To determine the FIFO channels between the processes, we may basically perform a standard array data flow analysis [30]. That is, for each execution of a read operation of a given data element in a function call, we need to find the source of the data, i.e., the corresponding write operation that wrote the data element. However, to reduce the communication of FIFO channels between different processes, in contrast to the standard data flow analysis and in contrast to [10] and [11], we also consider all the previous read operations from the same function call as possible sources of the data. That is why we call our approach a modified array data-flow analysis. The problem to be solved is then: given a read from an array element, what was the last write to or read (from that function call) from that array element? The last iteration of a function call satisfying some constraints can be obtained using Parametric Integer Programming (PIP) [31], where we compute the lexicographical *maximum* of the write (or read) source operations in terms of the iterators of the "sink" read operation. Since there may be multiple function calls that are potential sources of the data, and since we also need to express that the source operation is executed before the read (which is not a linear constraint, but rather a disjunction of n linear constraints, where n is the shared nesting level), we actually need to perform a number of PIP invocations.

For example, the first read access in function call F2 of the program fragment in Fig. 5(a) reads the data written by the function call F1, which results in a FIFO channel from process "F1" to process "F2," i.e., channel b in Fig. 5(b). In particular, the data flow from iteration i_w of function F1 to iteration $i_r = i_w$ of function F2. This information is captured by the integer relation

$$D_{\mathrm{F1}\to\mathrm{F2}} = \{(i_w, i_r) | i_r = i_w \wedge 0 \leq i_r \leq N-1\}. \quad (1)$$

For the second read access in function call F2, after the elimination of the temporary variable tmp, the data have already been read by the same function call after it was written. This results in a self-loop channel b_1 from F2 to itself described as

$$D_{\mathrm{F2}\to\mathrm{F2}} = \{(i_w, i_r) | i_w = i_r - 1 \wedge 1 \leq i_r \leq N-1\} \\ \cup \{(i_w, i_r) | i_w = i_r = 0\}. \quad (2)$$

In general, we obtain pairs of write/read and read operations such that some data flow from the write/read operation to the (other) read operation. These pairs correspond to the channels in our process network. For each of these pairs, we further obtain a union of integer relations

$$\bigcup_{j=1}^{m} D_j(\vec{i}_w, \vec{i}_r) \subset \mathbb{Z}^{n_1} \times \mathbb{Z}^{n_2} \quad (3)$$

where n_1 and n_2 are the numbers of loops enclosing the write and read operations, respectively, that connect the specific iterations of the write/read and read operations such that the first is the source of the second. As such, each iteration of a given read operation is uniquely paired off to some write or read operation iteration.

B. Computing FIFO Channel Sizes

Computing minimal deadlock-free buffer sizes is a nontrivial global optimization problem. This problem becomes easier if we first compute a deadlock-free schedule and then individually compute the buffer sizes for each channel. Note that this schedule is only computed for the purpose of computing the buffer sizes and is discarded afterward because the processes in our KPNs are self-scheduled due to the blocking read/write synchronization mechanism. The schedule we compute may not be optimal; however, our computations do ensure that a valid schedule exists for the computed buffer sizes. The schedule is computed using a greedy approach. This approach may not work for process networks in general, but since we only consider SANLPs, it does work for any KPN derived from SANLP. The basic idea is to place all iteration domains in a common iteration space at an offset that is computed by the scheduling algorithm. As in the individual iteration spaces, the execution order in this common iteration space is the lexicographical order. By fixing the offsets of the iteration domain in the common space, we have therefore fixed the relative order between any pair of iterations from any pair of iteration domains. The algorithm starts by computing for any pair of connected processes the minimal-dependence distance vector, i.e., a distance vector being the difference between a read operation and the corresponding write operation. Then, the processes are greedily combined, ensuring that all the minimal distance vectors are (lexicographically) positive. The end result is a schedule that ensures that every data element is written before it is read. For more information on this algorithm, we refer to [32], where it is applied to perform loop fusion on SANLPs.

After the scheduling, we may consider all FIFO channels to be self-loops of the common iteration space, and we can compute the buffer sizes with the following qualification: we will not be able to compute the absolute minimum buffer sizes, but at best the minimum buffer sizes for the computed schedule. To compute the buffer sizes, we compute the number of read iterations $R(i)$ that are executed before a given read operation i and subtract the resulting expression from the number of write iterations $W(i)$ that are executed before the given read operation, i.e.,

$$\#\text{elements in FIFO at operation } i: \; W(i) - R(i). \tag{4}$$

This computation can be entirely symbolically performed using the `barvinok` library [33] that efficiently computes the number of integer points in a parametric polytope. The result is a piecewise (quasi-)polynomial in the read iterators and parameters. The required buffer size is the maximum of this expression over all read iterations, i.e.,

$$\text{FIFO size} = \max\left(W(i) - R(i)\right). \tag{5}$$

To symbolically compute the maximum, we apply the Bernstein expansion [34] to obtain a parametric *upper bound* on the expression.

V. Automated Programming of MPSoCs

In this section, we present in detail our approach for the systematic and automated programming of MPSoCs synthesized with ESPAM. This approach is a main part of our design flow depicted in Fig. 1. For the sake of clarity, in this section, we explain the main steps in the ESPAM programming approach by going through an illustrative example. In the beginning, we give an example of the input specifications for ESPAM that describe an MPSoC. Next, from these example specifications, we show how the SW code for each processor in the MPSoC is generated, and we present our SW synchronization and communication primitives inserted in the code. Finally, we explain how the memory map of the MPSoC is generated.

A. Input Specification

The input to ESPAM consists of platform, application, and mapping specifications. An example of a platform specification is shown in Fig. 2(a). An example of an application depicted as a KPN is given in Fig. 8(a). It contains five processes communicating through seven FIFO channels. Part of the corresponding XML application specification for this KPN is shown in Fig. 6(a). Recall that this KPN in XML format is automatically generated by our translator tool (see Fig. 1) using the techniques presented in Section IV. For the sake of clarity, we only show the description of process Pr1 and channel FIFO1 in the XML code. Pr1 has one input port and one output port defined in lines 3–8. Pr1 executes a function

```
1  <application  name ="myKPN" >
   <process   name ="Pr1">
      <port  name = "p2"  direction = "in"  />
          <var  name = "in_0"  type = "myType"  />
5     </port>
      <port  name = "p1"  direction = "out" />
          <var  name = "out_0" type = "myType" />
      </port>

10    <process_code     name = "compute" >
          <arg   name = "in_0"  type = "input"  />
          <arg   name = "out_0" type = "output" />
          <loop   index = "k"  parameter = "N" >
             <loop_bounds  matrix = "[1,  1,0,-2;"
15                                     1,-1,2,-1]"/>
             <par_bounds  matrix = "[1,0,-1,384;"
                                     1,0,  1, -3]"/>
          </loop>
      </process_code>
20 </process>
          <!-- other processes omitted -->

   <channel name = FIFO1    size = "1">
      <fromPort name = "p1" />
25    <fromProcess name = "Pr1" />
      <toPort  name = "p4" />
      <toProcess name = "Pr3" />
   </channel>
          <!-- other channels omitted -->
30 </application>
```

(a)

```
1  <mapping name = "myMapping">

      <processor name = "uP1"   >
           <process  name = "Pr4"  />
5     </processor>

      <processor name = "uP2"   >
           <process  name = "Pr2" />
           <process  name = "Pr5" />
10    </processor>

      <processor name = "uP3"   >
            <proces   name = "Pr3" />
      </processor>
15
      <processor name = "uP4"   >
           <process  name = "Pr1" />
      </processor>

20 </mapping>
```

(b)

Fig. 6. Examples of application and mapping specifications. (a) Application specification. (b) Mapping specification.

called *compute* (line 10). The function has one input argument (line 11) and one output argument (line 12). There is a strong relation between the function arguments and the ports of a process given in lines 4 and 7. Lines 23–28 show an example of how the topology of a KPN is specified: FIFO1 connects processes Pr1 and Pr3 through ports p1 and p4. An example of a mapping specification is shown in Fig. 6(b), where process Pr4 is mapped onto processor uP1 (see lines 3–5), processes Pr2 and Pr5 are mapped onto processor uP2 (lines 7–10), etc. Notice that the mapping of channels to CMs is not specified. This mapping is related to the way processes are mapped to processors. Therefore, the mapping of channels to CMs cannot be arbitrary. It is automatically performed by ESPAM. This is described in Section V-D.

B. Code Generation for Processors

ESPAM uses the initial sequential application program, the corresponding KPN application specification, and the mapping specification to automatically generate the software (C/C++) code for each processor. The code for a processor contains a *control* code and a *computation* code. The *computation* code transforms the data that have to be processed by a processor. ESPAM directly extracts this code from the initial sequential program. The *control* code (*for* loops, *if* statements, etc.) determines the control flow, i.e., when and how many times data reading and data writing have to be performed by a processor as well as when and how many times the *computation* code has to be executed in a processor. The *control* code of a processor is generated by ESPAM according to the KPN application specification and the mapping specification.

In our example, process Pr1 is mapped onto processor uP4 (Fig. 6(b), lines 16–18). Therefore, ESPAM uses the XML specification of process Pr1, as shown in Fig. 6(a), to generate the *control* C code for processor uP4, as depicted in Fig. 7(a). In lines 4–7, the type of data transferred through the FIFO channels is declared. The data type can be a scalar or more complex type (in the example, it is a structure of one Boolean variable

```
 1 #include "primitives.h"
   #include "memoryMap.h"

    struct  myType  {
 5      bool flag;
        int    data[64];
   };
   int  N = 384;

10 void  main() {
       myType in_0;
       myType out_0;

       for ( int k=2; k<=2*N−1; k++ ) {
15         read( p2, &in_0, sizeof(myType) );
           compute( in_0, &out_0 );
           write( p1, &out_0, sizeof(myType) );
       }
19 }
```

(a)

```
 1 #ifndef  _PRIMITIVES_H_
   #define _PRIMITIVES_H_
   void  read( byte* port, void* data, int length )
       int  *isEmpty = port + 1;
 5     for ( int i=o; i<length; i++ ) {
           // reading is blocked if a FIFO is empty
           while ( *isEmpty ) {  }
           (byte* data)[i] = *port; // read from a FIFO
       }
10 }
   void  write( byte* port, void* data, int length )
       int  *isFull = port + 1;
       for ( int i=o; i<length; i++ ) {
           // writing is blocked if a FIFO is full
15         while ( *isFull ) {  }
           *port = (byte* data)[i]; // write to a FIFO
       }
   }
19 #endif
```

(b)

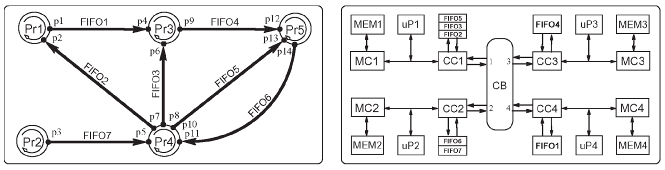
```
 1 #ifndef   _MEMORYMAP_H_
   #define   _MEMORYMAP_H_

   #define   p1    0xe0000008 //write addr. FIFO1
 5 #define   p4    0x00040001 //read addr. FIFO1
   #define   p7    0xe0000008 //write addr. FIFO2
   #define   p2    0x00010001 //read addr. FIFO2
   #define   p8    0xe0000010 //write addr. FIFO3
   #define   p6    0x00010002 //read addr. FIFO3
10 #define   p9    0xe0000008 //write addr. FIFO4
   #define   p12   0x00030001 //read addr. FIFO4
   #define   p10   0xe0000018 //write addr. FIFO5
   #define   p13   0x00010003 //read addr. FIFO5
   #define   p14   0xe0000008 //write addr. FIFO6
15 #define   p11   0x00020001 //read addr. FIFO6
   #define   p3    0xe0000010 //write addr. FIFO7
   #define   p5    0x00020002 //read addr. FIFO7

19 #endif
```

(c)

Fig. 7. Source code generated by ESPAM. (a) uP4.c. (b) primitives.h. (c) memoryMap.h.



(a) (b)

Fig. 8. Mapping example. (a) KPN. (b) Example platform.

and a 64-element array of integers as declared in the initial sequential program). There is one parameter (see Fig. 6(a), line 13) that has to be declared as well. This is done in line 8 of Fig. 7(a). Then, in lines 10–19 in the same figure, the behavior of processor uP4 is described. In accordance with the XML specification of process Pr1 in Fig. 6(a), the function *compute* is executed $2 * N - 2$ times. Therefore, a *for* loop is generated in the *main* routine for processor uP4 in lines 14–18 of Fig. 7(a). The *computation* code in function *compute* is extracted from the initial sequential program. This code is not important for our example; thus, it is not given here for the sake of brevity. The function *compute* uses the local variables in_0 and out_0 declared in lines 11 and 12 of Fig. 7(a). The input data come from FIFO2 through port p2, and the results are written to FIFO1 through port p1 [see Fig. 8(a)]. Therefore, before the function call, ESPAM inserts a *read* primitive to read from FIFO2 that initializes variable in_0. After the function call, ESPAM inserts a *write* primitive to send the results (the value of variable out_0) to FIFO1, as shown in Fig. 7(a) (lines 15 and 17). When several processes are mapped onto one processor, a schedule is required in order to guarantee a proper execution order of these processes. The ESPAM tool automatically finds a local static schedule based on the grouping technique for the processes presented in [35].

C. SW Communication and Synchronization Primitives

The code of the *read*/*write* communication and synchronization primitives [used in lines 15 and 17 in Fig. 7(a)] is shown in Fig. 7(b). This code is automatically generated by ESPAM. Both primitives have three parameters, i.e., *port*, *data*, and *length*. The parameter *port* is the address of the memory location through which a processor can access a given FIFO channel for reading/writing. The parameter *data* is a pointer to a local variable, and the parameter *length* specifies the amount of data (in bytes) to be moved from/to the local variable to/from the FIFO channel. Recall that a FIFO channel is seen by a processor as two memory locations in its CM address space. A processor uses the first location to read/write data from/to the FIFO channel, and the second location is used to read the empty/full status of the channel. The primitives implement the blocking synchronization mechanism between the processors in the following way: First, the status of a channel that has to be read/written is checked. A channel status is accessed using the locations defined in lines 4 and 12. The blocking is implemented by while loops with empty bodies in lines 7 and 15. A loop iterates (does nothing) while a channel is full or empty. Then, in lines 8 and 16, the actual data transfer is performed.

D. Memory Map Generation

Each FIFO channel in our MPSoCs has separate read and write ports. A processor accesses a FIFO for read operations using the *read* synchronization primitive (as described in Section V-C). The parameter *port* specifies the address of the read port of the FIFO channel to be accessed. In the same way, the processor writes to a FIFO using the write synchronization primitive where the parameter *port* specifies the address of the write port of this FIFO. The FIFO channels are implemented in the CMs (see Section III); therefore, the addresses of the FIFO ports are located in the processors' address space where the CM segment is defined. The memory map of an MPSoC generated by ESPAM contains the values that define the read and write addresses of each FIFO channel in the system.

The first step in the memory map generation is the mapping of the FIFO channels in the KPN application specification onto the CMs in the multiprocessor platform. This mapping cannot be arbitrary. ESPAM maps FIFO channels onto the CMs of the processors in the following automated way. First, for each process in the application specification, ESPAM finds all the channels this process writes to. Then, from the mapping specification, ESPAM finds which processor corresponds to the current process and maps the found channels in the processor's local CM. For example, consider the mapping specification shown in Fig. 6(b), which defines only the mapping of the processes of the KPN in Fig. 8(a) to the processors in the platform shown in Fig. 8(b). Based on this mapping specification, ESPAM automatically maps FIFO2, FIFO3, and FIFO5 onto the CM of processor uP1 because process Pr4 is mapped onto processor uP1, and process Pr4 writes to channels FIFO2, FIFO3, and FIFO5. Similarly, FIFO4 is mapped onto the CM of processor uP3, and FIFO1 is mapped onto the CM of uP4. Since both processes Pr2 and Pr5 are mapped onto processor uP2, ESPAM maps FIFO6 and FIFO7 onto the CM of this processor.

After the mapping of the channels onto the CMs, ESPAM generates the memory map of MPSoC, i.e., it generates values for FIFOs' read and write addresses. For the mapping example illustrated in Fig. 8(b), the generated memory map is shown in Fig. 7(c). Notice that FIFO1, FIFO2, FIFO4, and FIFO6 have equal write addresses (see lines 4, 6, 10, and 14). This is not a problem because writing to these FIFOs is done by different processors, and these FIFOs are located in the local CMs of these different processors, i.e., these addresses are local processor write addresses. The same applies for the write addresses of FIFO3 and FIFO7. However, as explained in Section III, all processors can read from all the FIFOs via a communication component. Therefore, the read addresses have to be unique in the MPSoC memory map, and the read addresses have to precisely specify the CM in which a FIFO is located. To accomplish this, a read address of a FIFO has two fields, i.e., a CM number and a FIFO number within a CM.

Consider, for example, FIFO3 in Fig. 8(b). It is the second FIFO in the CM of processor uP1; thus, this FIFO is numbered with 0002 in this CM. In addition, the CM of uP1 can be accessed for reading through port 1 of the communication component CB, as shown in Fig. 8(b); thus, this CM is uniquely numbered with 0001. As a consequence, the unique read address of FIFO3 is determined to be 0x00010002—see line 9 in Fig. 7(c), where the first field 0001 is the CM number, and the second field 0002 is the FIFO number in this CM. In the same way, ESPAM automatically determines the unique read addresses of the rest of the FIFOs that are listed in Fig. 7(c).

VI. Experiments and Results

To show the effectiveness of our approach, in this section, we present three experiments and the results we have obtained by implementing and executing three image processing applications onto several MPSoCs using our system design flow and the ESPAM tool presented in Section II. These applications are a Sobel edge detection, a Discrete Wavelet Transform (DWT), and a Motion JPEG (M-JPEG) encoder.

A. Experiment 1

In this experiment, we used our design flow to design and program multiprocessor systems that execute the Sobel, DWT, and M-JPEG applications. We started from SANLPs written in C, and automatically derived the parallel application specifications (KPNs) using our translator tool [9]. For the Sobel and DWT applications, we used platform specifications with three MicroBlaze processors connected P2P, and for the M-JPEG application, a platform specification consisting of four MicroBlaze processors was used, again connected P2P. To show the achieved speedup, we compare the performance of the automatically generated MPSoCs with the performance of single-processor systems, one MicroBlaze or one PowerPC, that execute the initial C programs for the Sobel, DWT, and M-JPEG applications. The single-processor systems and the C code were designed and optimized by hand. The achieved performance results for processing one image frame with size 128 × 128 pixels are shown in Fig. 9. The first (left most) bar for each application shows the performance of the application executed on a single MicroBlaze processor, the second bar shows the performance of a single PowerPC system, and the third (right most) bar shows the performance of our MPSoCs with several (three or four) MicroBlaze processors, as described above. The achieved speedup by our multiprocessor systems compared to the single-processor systems is application dependent. The MPSoC for Sobel achieves a speedup of 2.2× compared to the single MicroBlaze system, the MPSoC for DWT achieves a speedup of 2.1×, and the MPSoC for M-JPEG achieves a speedup of 3.75× compared to one MicroBlaze. The speedup is influenced by the data dependences between different processes and the communication/computation ratio in the KPNs that specify the applications. The dependences among the processes in the three KPNs form an acyclic graph, which allows all the processors to operate in parallel in a pipeline fashion. Therefore, the achieved speedup is mainly determined by the communication/computation ratio. The communication in the Sobel and DWT systems takes more than 50% of the total execution time because the processes executed on the processors are too fine grained. In contrast, the processes in M-JPEG are coarse grained; therefore, the processors in the M-JPEG system spend no more than 10% of the execution time for communication. As a result, the M-JPEG system achieved a higher speedup.

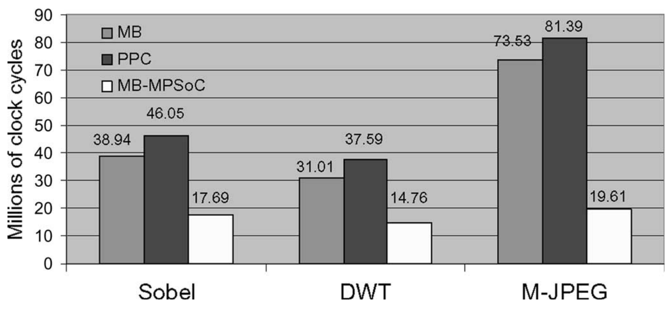


Fig. 9. Performance results.

B. Experiment 2

The main objective of this experiment is to evaluate the effectiveness of our design flow for automated MPSoC synthesis, programming, and implementation in terms of total design time, i.e., how fast alternative multiprocessor systems can be synthesized, programmed, and implemented. In addition, we validate in terms of accuracy the high-level simulation models used in SESAME [5] to explore the design space. We present a comparison between the results obtained by running system-level simulations (during design space exploration) and real implementations of the M-JPEG encoder application.

As explained in Section II, ESPAM needs three input specifications, namely an application specified as KPN, a platform specification, and a mapping specification. Table I shows that the KPN specification of the M-JPEG application was derived in 22 s from a sequential C code using our translator tool [9].

TABLE I
PROCESSING TIMES (IN HOURS: MINUTES: SECONDS)

	KPN Deriv.	Plat./Map. Deriv.	System to RTL conv.	Physical Implement.	Manual Modific.
Translator	00:00:22	—	—	—	00:30:00
SESAME	—	01:26:00	—	—	—
ESPAM	—	—	00:25:00	—	—
XPS	—	—	—	18:29:00	—

A small manual modification (taking no longer than 30 min) to the initial C code was necessary to meet the translator tool input requirements. Generating the KPN specification is a one-time effort since the same specification is used for all the subsequent exploration and implementation steps.

The platform and the mapping specifications were generated by SESAME after performing design space exploration using the derived KPN specification of the M-JPEG application as an input. We explored heterogeneous CB-based MPSoC platforms with up to four processors (MicroBlaze or PowerPC). In our design space exploration, we used three degrees of freedom, namely the number of processors in the platform (1 to 4), the type of processors (MicroBlaze versus PowerPC), and the mapping of application processes onto the processors. Because of SESAME's efficiency and high level of abstraction modeling, we were able to exhaustively explore the resulting design space—which consists of 10 148 design points—using system-level simulations. As can be seen in Table I, this design space sweep took 1.5 h.

We selected 11 design points out of 10 148 that represent 11 alternative MPSoC architectures with the best found application-to-architecture mappings in terms of performance of the application executed on these MPSoCs. The SESAME tool generated the platform and mapping specifications for each of the selected 11 design points. Having these specifications, together with the application specification (KPN), our ESPAM tool synthesized, programmed, and generated 11 multiprocessor systems at RTL in 25 min (see Table I). The generated files were automatically imported to the Xilinx Platform Studio (XPS) tool [21] for physical implementation, i.e., mapping, place, and route onto the FPGA. For prototyping in this experiment, we used an FPGA board with the Xilinx VirtexII-Pro-20 device. It took the XPS tool more than 18 h to implement the 11 MPSoCs. All tools ran on a Pentium-IV machine at 1.8 GHz with 1 GB of RAM. The figures in Table I show that a complete implementation and programming of all the 11 platforms starting from high-abstraction system-level specifications can be obtained in about 22 h using our system design flow. So, a significant reduction of design time is achieved. For comparison, we refer to [36], where the same M-JPEG encoder application was used in a case study. In the presented work, the authors report that only a manual partitioning of this application took three months. The performance results of our second experiment are shown in Fig. 10. The performance numbers obtained during design space exploration by simulation of the system-level models for the selected MPSoCs are shown in Fig. 10(a). The real performance numbers for the same MPSoCs implemented and run on FPGA are shown in Fig. 10(b). In both figures, the left axis shows the performance numbers (in clock cycles) of each alternative MPSoC. The right axis shows

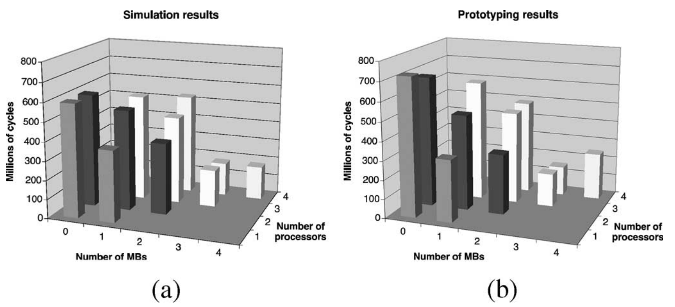


Fig. 10. Validation experiment: simulation results versus actual measurements. (a) Simulation results (SESAME). (b) Prototyping results (ESPAM).

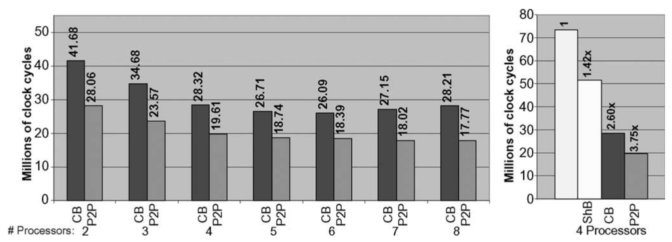


Fig. 11. Performance results.

how many processors an MPSoC contains, and the bottom axis shows how many of them are *MicroBlaze* processors. For example, the bar with right coordinate 4 and bottom coordinate 2 (4,2) represents the performance of a system that has four processors, where two of them are *MicroBlazes*. It means that the other two are *PowerPC* processors. The empty points in the figures represent multiprocessor systems that cannot be built in general, i.e., point (3,4) means a system with three processors and four of them to be *MicroBlazes*.

Comparing the simulation numbers with the implementation numbers in Fig. 10, we see that the system-level simulations adequately show the correct performance trends with an average error of 13% and worst-case error of 28%. The inaccuracies in terms of the absolute cycle numbers are mainly caused by the high-level modeling of the processors' behavior. Indeed, this is the price we have to pay in order to achieve very fast simulations and design space exploration.

C. Experiment 3

In this experiment, we used ESPAM to implement the M-JPEG application onto 16 alternative homogeneous MPSoCs containing up to eight *MicroBlaze* processors connected through a CB or P2P. The main objective of this experiment is to evaluate the efficiency of the automatically generated MPSoC implementations in terms of performance and resource utilization.

1) Performance Results: The MPSoCs contain up to eight *MicroBlaze* processors connected either P2P or through a CB. The performance numbers, depicted in the left part of Fig. 11, indicate that for the M-JPEG application and the alternative platforms and mappings we have experimented with, it is not reasonable to go beyond four processors. However, in this experiment, we also wanted to check the system complexity that can be reached by using our approach and the ESPAM tool.

TABLE II
RESOURCE UTILIZATION

	#Slices	#4-Input LUT	#Flip-Flops	#BRAMs
4 Proc. Shared Bus	3640 (39%)	4722 (25%)	2354 (12%)	85 (60%)
4 Proc. Crossbar	3653 (39%)	4748 (25%)	2357 (12%)	85 (60%)
4 Proc P2P System	3263 (35%)	3929 (21%)	2405 (12%)	88 (62%)
4 CCs	288 (2%)	468 (2%)	116 (1%)	—
4 Port CB	397 (3%)	587 (3%)	56 (1%)	—
4 Port Bus	366 (3%)	541 (2%)	47 (1%)	—

We were able to implement and program systems containing up to eight *MicroBlaze* processors. The number of the processors is only limited by the size of the FPGA chip we used for prototyping, i.e., a Xilinx VirtexII-6000 FPGA.

We comment below on the performance of the systems containing four *MicroBlaze* processors. The performance speedup achieved by these systems is depicted in the right part of Fig. 11. The leftmost bar shows the performance of the M-JPEG application run on one *MicroBlaze* from our first experiment. We use this as a reference number. We achieved a performance speedup of 2.60× for the system with four processors and a CB component, and for the system with four processors and P2P connections, the performance speedup is 3.75×, whereas the theoretical maximum is 4×. For comparison, we also implemented a multiprocessor platform with four processors connected through a ShB with round-robin arbitration policy. The achieved speedup by this multiprocessor system (the second bar) is only 1.42×. This clearly shows that a ShB architecture is not an efficient architecture for building high-performance multiprocessor systems.

Using ESPAM, we implemented, ran, and measured the performance of the 16 alternative MPSoCs described above in approximately two days. This fact indicates that in a relatively short amount of time, we were able to explore the performance of alternative multiprocessor systems through real implementations and measurements of actual numbers. These numbers are 100% accurate. Gathering these numbers is faster than running cycle-accurate simulations of the MPSoCs. We do not know how much time is needed for an experienced designer to verify an RTL simulation of several hardwired components and several processors running in parallel and executing different programs. However, we know that only setting up and performing such simulation may take days. Of course, performing simulation at a higher level of abstraction is faster than the implementation and measurement of the real performance (see Experiment 2), but the 100% accuracy of the numbers cannot be achieved as we showed in Experiment 2. Therefore, in our design flow, we apply the following strategy when designing an MPSoC: 1) We perform simulation at a high level of abstraction in order to do very fast design space exploration and to narrow down the design space to a few design points, for example, 20. 2) We perform 100% accurate exploration in the narrowed design space by real MPSoC implementations and measurements of the actual numbers in order to select the best MPSoC implementation.

2) Synthesis Results: In Table II, we present the overall resource utilization of the multiprocessor systems with four processors that we considered in this experiment. We also present the utilization results for the CCs, a four-port CB component, and a four-port ShB component (BUS). The FPGA resources are grouped into slices that contain four-input lookup tables and flip-flops. The first three rows in the table show that the multiprocessor systems utilize around 40% of the slices in the FPGA. In addition, the last three rows show that our communication component (CB or BUS) together with the CCs in each system utilize a minor portion of the FPGA slices—only 5%. These numbers clearly indicate that our approach to connect processors through communication components and CMs is very efficient in terms of slice utilization. The last column in Table II shows a relatively high utilization (61%) of the on-chip memory. This high utilization is not related to inefficiency in our approach to connect processors via CMs because for each M-JPEG system we use a maximum of nine BRAM blocks to implement FIFO buffers distributed over four CMs. The high BRAM utilization is due to the fact that the M-JPEG is a relatively complex application, and almost all BRAM blocks are used for the program and data memory of the four microprocessors in our platforms.

VII. CONCLUSION

In this paper, we have presented our system design methods and techniques that are implemented in the ESPAM tool for automated multiprocessor platform synthesis, implementation, and programming. This automation significantly reduces the design time starting from system-level specification and going down to complete implementation. Based on our experiments, we conclude that our ESPAM tool, together with the translator tool [9], SESAME [5], and the XPS tool [21], is able to systematically, automatically, and quickly implement and program a multiprocessor platform—within 2 h. The results presented in this paper show that our approach of connecting processors through CCs and CMs is efficient in terms of HW utilization and performance speedup. For an M-JPEG encoder application implemented with four processors, the communication logic utilizes only 5% of the resources. We achieved a speedup close to the theoretical maximum (4×) as compared to a single-processor system.

REFERENCES

[1] G. Martin, "Overview of the MPSoC design challenge," in *Proc. DAC*, Jul. 2006, pp. 274–279.

[2] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. IFIP Congr.*, 1974, pp. 471–475.

[3] T. Stefanov *et al.*, "System design using Kahn process networks: The Compaan/Laura approach," in *Proc. DATE*, Feb. 2004, pp. 340–345.

[4] A. Nieuwland *et al.*, *C-HEAP: A Heterogeneous Multi-Processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems*. Norwell, MA: Kluwer, 2002.

[5] A. Pimentel *et al.*, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 99–112, Feb. 2006.

[6] E. de Kock, "Multiprocessor mapping of process networks: A JPEG decoding case study," in *Proc. ISSS*, Oct. 2002, pp. 68–73.

[7] K. Goossens *et al.*, "Guaranteeing the quality of services in networks on chip," in *Networks on Chip*. Norwell, MA: Kluwer, 2003, pp. 61–82.

[8] B. Dwivedi *et al.*, "Automatic synthesis of system on chip multiprocessor architectures for process networks," in *Proc. CODES+ISSS*, Sep. 2004, pp. 60–65.

[9] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: A tool for improved derivation of process networks," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, p. 19, Jan. 2007. Article ID 75947, 13 pages, DOI:10.1155/2007/75947.

[10] A. Turjan *et al.*, "Translating affine nested-loop programs to process networks," in *Proc. CASES*, Sep. 2004, pp. 220–229.
[11] B. Kienhuis *et al.*, "Compaan: Deriving process networks from Matlab for embedded signal processing architectures," in *Proc. CODES*, May 2000, pp. 13–17.
[12] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere, "LAURA: Leiden architecture research and exploration tool," in *Proc. FPL*, Sep. 2003, pp. 911–920.
[13] M. J. Rutten *et al.*, "A heterogeneous multiprocessor architecture for flexible media processing," *IEEE Des. Test Comput.*, vol. 19, no. 4, pp. 39–50, Jul./Aug. 2002.
[14] A. Gerstlauer and D. Gajski, "System-level abstraction semantics," in *Proc. ISSS*, Oct. 2002, pp. 231–236.
[15] A. Jerraya *et al.*, "Programming models and HW–SW interfaces abstraction for multi-processor SoC," in *Proc. DAC*, Jul. 2006, pp. 280–285.
[16] D. Lyonnard *et al.*, "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip," in *Proc. DAC*, Jun. 2001, pp. 518–523.
[17] L. Gauthier, S. Yoo, and A. Jerraya, "Automatic generation and targeting of application specific operating systems and embedded systems software," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 11, pp. 1293–1301, Nov. 2001.
[18] P. Paulin *et al.*, "Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 7, pp. 667–680, Jul. 2006.
[19] H. Yu, R. Domer, and D. Gajski, "Embedded software generation from system level design languages," in *Proc. ASP-DAC*, Jan. 2004, pp. 463–468.
[20] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink, "Design and programming of embedded multiprocessors: An interface-centric approach," in *Proc. CODES+ISSS*, Sep. 2004, pp. 206–217.
[21] Xilinx, Inc., *Xilinx Platform Studio and the Embedded Development Kit*. EDK version 8.1i edition. [Online]. Available: www.xilinx.com/ise/embedded_design_prod/platform_studio.htm
[22] Altera, Inc., (2005, Dec.). *Quartus II Handbook Volume 4: SOPC Builder*. [Online]. Available: www.altera.com/literature/quartus2/lit-qts-sopc.jsp
[23] *Multiprocessor Solutions With FPGAs*. (2005). White paper, FPGA and Programmable Logic Journal. [Online]. Available: www.fpgajournal.com/whitepapers_2005/altera_20050224.htm
[24] T. Parks, "Bounded scheduling of process networks," Ph.D. dissertation, Univ. California, EECS Dept., Berkeley, 1995. Tech. Rep. UCB/ERL-95-105.
[25] J. Buck and E. Lee, "Scheduling dynamic data flow graphs with bounded memory using the token flow model," in *Proc. IEEE Conf. Acoust., Speech, Signal Process.*, Apr. 1993, pp. 429–432.
[26] M. Geilen and T. Basten, *Requirements on the Execution of Kahn Process Networks*, vol. 2618. Berlin, Germany: Springer-Verlag, 2003, pp. 319–334.
[27] E. de Kock *et al.*, "YAPI: Application modeling for signal processing systems," in *Proc. DAC*, Jun. 2000, pp. 402–405.
[28] E. A. Lee *et al.*, "Ptolemy II: Heterogeneous concurrent modeling and design in Java," Univ. California at Berkeley, Berkeley, Tech. Rep. UCB/ERL-M99/40, 1999.
[29] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model*, vol. 1132. Berlin, Germany: Springer-Verlag, 1996, pp. 79–103.
[30] P. Feautrier, "Dataflow analysis of scalar and array references," *Int. J. Parallel Program.*, vol. 20, no. 1, pp. 23–53, Feb. 1991.
[31] P. Feautrier, "Parametric integer programming," *Oper. Res.*, vol. 22, no. 3, pp. 243–268, Sep. 1988.
[32] S. Verdoolaege *et al.*, "Multi-dimensional incremental loop fusion for data locality," in *Proc. ASAP*, Jun. 2003, pp. 17–27.
[33] S. Verdoolaege *et al.*, "Analytical computation of Ehrhart polynomials: Enabling more compiler analyses and optimizations," in *Proc. CASES*, Sep. 2004, pp. 248–258.
[34] P. Clauss *et al.*, "Symbolic polynomial maximization over convex sets and its application to memory requirement estimation," Université Louis Pasteur, Strasbourg, France, ICPS Res. Rep. 06-04, Oct. 2006.
[35] T. Stefanov and E. Deprettere, "Deriving process networks from weakly dynamic applications in system-level design," in *Proc. CODES+ISSS*, Oct. 2003, pp. 90–96.
[36] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere, "System level design with SPADE: An M-JPEG case study," in *Proc. ICCAD*, Nov. 2001, pp. 31–38.

Hristo Nikolov (S'05) received the Dipl.Ing. and M.S. degrees in computer engineering from The Technical University of Sofia, Sofia, Bulgaria, in 2001. He is currently working toward the Ph.D. degree in computer science in the Leiden Embedded Research Center, Leiden University, Leiden, The Netherlands.

From 2001 to 2004, he was a Research and Development Engineer with Fabless, Ltd., Sofia. Since 2004, he has been with the Leiden Institute of Advanced Computer Science, Leiden University. His research interests are in system-level design automation, multiprocessor systems-on-chip design, and hardware/software codesign.

Todor Stefanov (S'01–M'05) received the Dipl.Ing. and M.S. degrees in computer engineering from The Technical University of Sofia, Sofia, Bulgaria, in 1998 and the Ph.D. degree in computer science from Leiden University, Leiden, The Netherlands, in 2004.

From 1998 to 2000, he was a Research and Development Engineer with Innovative Micro Systems, Ltd., Sofia. Since 2000, he has been with the Leiden Institute of Advanced Computer Science, Leiden University, where he was a Research Assistant and is currently a PostDoc Researcher at the Leiden Embedded Research Center. His research interests include several aspects of embedded systems design, with particular emphasis on system-level design automation, multiprocessor systems-on-chip design, and hardware/software codesign.

Ed Deprettere (M'83–SM'88–F'96) received the M.S. degree from the University of Ghent, Ghent, Belgium, in 1968 and the Ph.D. degree from Delft University of Technology, Delft, The Netherlands, in 1981.

From 1980 to 1999, he was a Professor with the Department of Electrical Engineering, Circuits and Systems Section, Signal Processing Group. Since January 1, 2000, he has been a Professor with the Leiden Institute of Advanced Computer Sciences, Leiden University, Leiden, The Netherlands, where he is also the Head of the Leiden Embedded Research Center. He is the Editor and co-editor of four books and several special issues of international journals. He is on the editorial board of three journals. His current research interests are in the system-level design of embedded systems, in particular for signal, image, and video processing applications.