Multi-processor System Design with ESPAM

Hristo Nikolov Todor Stefanov Ed Deprettere
Leiden Institute of Advanced Computer Science
Leiden University, The Netherlands
{nikolov,stefanov,edd}@liacs.nl

ABSTRACT

For modern embedded systems, 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 these platforms. Unfortunately, most of the current design methodologies and tools are based on Register Transfer Level (RTL) descriptions, mostly created by hand. Such methodologies are inadequate, because creating RTL descriptions of complex multiprocessor systems is error-prone and time consuming.

As an efficient solution to these two problems, in this paper we propose a methodology and techniques implemented in a tool called ESPAM *for automated multiprocessor system design and implementation.* ESPAM *moves the design specification from RTL to a higher, so called system level of abstraction. We explain how starting from system level platform, application, and mapping specifications, a multiprocessor platform is synthesized and programmed in a systematic and automated way. Furthermore, we present some results obtained by applying our methodology and* ESPAM *tool to automatically generate multiprocessor systems that execute a real-life application, namely a Motion-JPEG encoder.*

Categories and Subject Descriptors: J.6 [Computer-aided engineering]: Computer-aided design (CAD).

General Terms: Algorithms, Design, Experimentation.

Keywords: System-Level Design, Heterogeneous MPSoCs, Kahn Process Networks.

1. INTRODUCTION

Moore's law predicts exponential growth over time of the number of transistors that can be integrated in a single chip. The intrinsic computational power of a chip must not only be used efficiently and effectively, but also the time and effort to design a system containing both hardware and software must remain acceptable. Unfortunately, current system design methodologies (including platform design and application specification) are still based on Register Transfer Level (RTL) platform/application descriptions created by hand using, for example, VHDL and/or C. Such methodologies were effective in the past. However, applications and platforms used in many of today's new system designs are so complex that traditional design practices are now inadequate, because creating RTL descriptions of complex multiprocessor systems is error-prone and time-consuming. Moreover, 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. Furthermore, using traditional logic simulation to verify a large design represented in RTL is computationally expensive and extremely slow.

CODES+ISSS'06, October 22–25, 2006, Seoul, Korea.

1.1 Problem Description

For all the reasons stated above, we conclude that the use of a 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 which we call *Implementation Gap*. Indeed, on the one hand, the RTL system specification is very detailed and close to an implementation, thereby allowing 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, but currently we 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 convert effectively and efficiently a system level specification to a RTL level specification.

1.2 Paper Contributions

In this paper we present our tool ESPAM (Embedded System-level Platform synthesis and Application Mapping) that implements our methods and techniques for systematic and automated multiprocessor platform implementation and programming. They successfully bridge the gap between the *system level* specification and the *RTL level* specification which we consider as the main contribution of this paper. More specifically, ESPAM allows a system designer to specify a multiprocessor system at a high level of abstraction in a short amount of time. Then ESPAM refines this specification to a real implementation in a systematic and automated way thereby successfully closing the implementation gap mentioned earlier. This reduces the design time from months to hours. As a consequence, a very accurate exploration of the performance of alternative multiprocessor platforms becomes feasible at implementation level in a few hours.

The success of our methods and techniques in closing the implementation gap is based on the underlying application model and system level platform model. ESPAM can implement data-flow dominated (streaming) applications onto multiprocessor platform instances efficiently and in an automated way. For the latter, a crucial role is played by the Kahn Process Network (KPN) [1] model of computation which we use as an application model. Many researchers [2] [3] [4] [5] [6] [7] have already indicated that KPNs are suitable for 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 which we call a platform model. We consider this an important contribution of this paper because our set of components (platform model) allows system designers to specify (construct) very fast and easily many alternative multiprocessor platforms that are systematically and automatically implemented and programmed by our tool ESPAM.

1.3 Related Work

Systematic and automated application-to-architecture mapping has been widely studied in the research community. The closest to our work is the Compaan/Laura design flow [2]. It uses KPN specifications for automated mapping of applications targeting FPGA implementations. The reported results are only for processor-coprocesor architectures whereas our ESPAM tool allows an automated implementation of KPN specifications onto multiprocessor platforms.

The Eclipse work [3] defines a scalable architecture template for designing stream-oriented multiprocessor SoCs using the KPN model of computation 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.

In [8] a design flow for the generation of application-specific multiprocessor architectures is presented. This work is similar to our approach in the sense that we also generate multiprocessor systems based on instantiation of generic parameterized architecture components where very efficient communication controllers are generated automatically to connect processors to communication networks. However, many steps of the design flow in [8] are performed manually. As a consequence a full implementation of a system with 4 processors connected point-to-point takes around 33 hours. In contrast, our design flow is fully automated and a full implementation of a system with 8 processors connected point-to-point or via crossbar or shared bus takes around 2 hours.

A system level semantics for a system design process formalization is presented in [9]. 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.

Companies such as Xilinx and Altera provide design tool chains attempting to generate efficient implementations starting from descriptions higher than (but still related to) the RTL level of abstraction. The required input specifications are so detailed that designing a single processor system is still error-prone and time consuming, let alone alternative multiprocessor systems. In contrast, our design methodology raises the design focus to an even higher level of abstraction allowing the design and the programming of multiprocessor systems in a short amount of time. Moreover, this does not sacrifice the possibility for automatic and systematic design implementation because our ESPAM tool supports it.

2. ESPAM DESIGN FLOW: OVERVIEW

In this section we give an overview of our system design methodology which is centered around our ESPAM tool developed to close the implementation gap described in Section 1.1. This is followed by a description of our system level platform model and platform synthesis in Section 3. In Section 4 we discuss the automated programming of multiprocessor platforms, and in Section 5 we present some results that we have obtained using ESPAM. Section 6 concludes the paper.

Our system design methodology is depicted as a design flow in Figure 1. There are three levels of specification in the flow. They are SYSTEM-LEVEL specification, RTL-LEVEL specification, and GATE-LEVEL specification. The SYSTEM-LEVEL specification consists of three parts: 1) *Platform Specification* describing the topology of a platform using our system level platform model, i.e., using generic parameterized system components; 2) *Application Specification* describing an application as a Kahn Process Network (KPN), i.e., network of concurrent processes communicating via FIFO channels. The KPN specification reveals the task-level parallelism available in the application; 3) *Mapping Specification* describing the relation between all processes and FIFO channels in *Application Specification* and all components in *Platform Specification*.

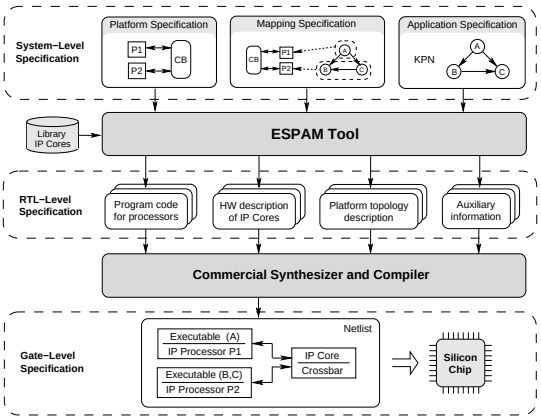


Figure 1: ESPAM System Design Flow.

The SYSTEM-LEVEL specification is given as input to ESPAM. First, ESPAM constructs a platform instance following 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 defines only the key system components of the platform and their attributes. Second, ESPAM refines the abstract platform model to an elaborate (detailed) parameterized RTL model which 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 program code for each processor in the multiprocessor platform in accordance with the application and mapping specifications.

The output of ESPAM, namely a 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 register transfer level. It consists of four parts: 1) *Platform topology* description defining in greater detail the multiprocessor platform; 2) *Hardware descriptions of IP cores* containing predefined and custom IP cores used in 1). ESPAM selects predefined IP cores (processors, memories, etc.) from *Library IP Cores*, see Figure 1. Also, it generates custom IP cores needed as a glue/interface logic between components in the platform; 3) *Program code for processors* — to execute the application on the synthesized multiprocessor platform, ESPAM generates program source code files for each processor in the platform. 4) *Auxiliary information* containing files which give tight control on the overall specifications, such as defining precise timing requirements and prioritizing signal constraints.

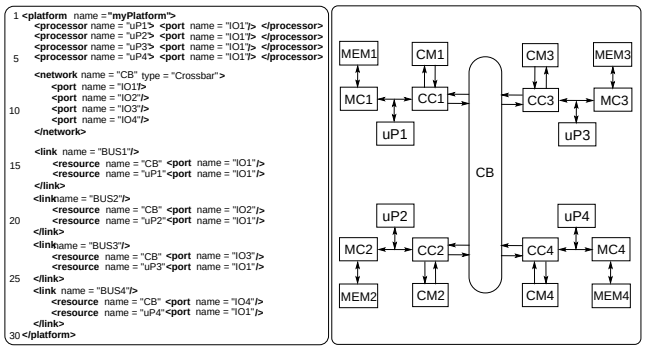
With the descriptions above, a commercial synthesizer can convert a RTL-LEVEL specification to a GATE-LEVEL specification, thereby generating the target platform gate level netlist, see the bottom part of Figure 1. This GATE-LEVEL specification is actually the system implementation. The current prototype version of ESPAM facilitates automated multiprocessor platform synthesis and programming using Xilinx VirtexII-Pro FPGAs. ESPAM uses the Xilinx Platform Studio (XPS) tool as a back-end to generate the final bit-stream file that configures a specific FPGA. We use the FPGA platform technology for

prototyping purposes only. Our ESPAM is general and flexible enough to be targeted to other physical platform technologies. A real-life industrially-relevant application, namely Motion-JPEG encoder, has been fully implemented onto several alternative multiprocessor platforms by using the ESPAM and XPS design tools.

3. PLATFORM MODEL AND SYNTHESIS

In our design methodology, the platform model is a library of generic parameterized components. In order to support 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 *Processing* components, *Memory* components, *Communication* components, *Communication Controller*, and *Links*. *Memory* components are used to specify the processors' local program and data memories and to specify data communication storages (buffers) between processors. Further we will call the data communication storages *Communication Memories*. We have developed a point-to-point network, a crossbar switch, and a shared bus component with several arbitration schemes (Round-Robin, Fixed Priority, and TDMA). These *Communication* components determine the communication network topology of a multiprocessor platform. The *Communication controller* 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 easily, simply by 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 Figure 2a. This specification, written in XML format, consists



a) Platform Specification **b)** The elaborate platform

Figure 2: Example of a Multiprocessor Platform.

of three parts which define processing components (four processors, lines 2-5), communication component (crossbar, lines 7-12), and links (lines 14-29). The links specify the connections of the processors to the communication component. To guarantee correct-by-construction automated platform synthesis and implementation, our ESPAM tool runs a consistency check on each platform specified by a designer. This includes finding impossible and/or meaningless connections between system level platform components as well as parameter values that are out of range. Notice that in the specification a designer does not have to take care of 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 the communication components. Second, it automatically attaches memories and memory controllers (MCs) to each processor. Third, the tool automatically synthesizes, instantiates, and connects all necessary communication memories (CMs) and communication controllers (CCs) to allow efficient and safe (lossless) data communication and synchronization between the components.

The elaborate platform generated by ESPAM is shown in Figure 2b. The processors (uPs) transfer data between each other through the CMs. A communication controller connects a communication memory to the data bus of the processor it belongs to and to a communication component. Since every programmable processor has a data bus, processors of different types can easily be connected into a heterogeneous multiprocessor platform by using our CCs. Each CC implements the processor's local bus-based access protocol to the CM for write operations and the access to the communication component (CB) for read operations. Each CM is organized as one or more FIFO buffers. We have chosen such organization because the inter-processor synchronization in the platform can be implemented in a very simple and efficient way by blocking read/write operations on empty/full FIFO buffers located in the communication memory. As a result, memory contention is avoided.

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 needed as well. At the same time, we want to stay as close as possible to the KPN semantics because it guarantees the highest possible communication performance. Therefore, in our approach each processor writes only to its local communication memory (CM) and uses the communication component only to read data from all other communication memories. This means that a processor can always write if there is room in its local CM (if this CM is large enough, the processor may never block on writing). A processor blocks when reading other processors' CMs if data is not available or the communication resource is currently not available.

3.1 Processing Components

In our approach we do not propose a design of processing components. Instead, we use 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 processing components include two programmable processors, namely $MicroBlaze$ (MB) and $PowerPC$ (PPC). Our platform model is general enough to be extended easily with additional (processing) components. Notice that only the processing components in our platform model are related to a particular technology (currently to Xilinx VirtexII-Pro FPGAs). All other components discussed in this section are technology independent.

3.2 Communication Memory Components

We implement the communication memories of a processor by using dual-port memories. Logically, a communication memory (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 inter-processor 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 the inter-processor synchronization. The multi-FIFO behavior of a communication memory is implemented by the communication controller described below. However, if a communication memory contains only one FIFO, we use a dedicated FIFO component which simplifies the structure of the communication controller.

3.3 Communication Controller

The structure of the Communication Controller (CC) is shown in Figure 3. It consists of two blocks, namely Interface Unit and FIFOs Unit. The Interface Unit contains an address decoder, fifos' control logic, and logic to generate read requests to the communication component. When a processor has to write data to its local Communica-

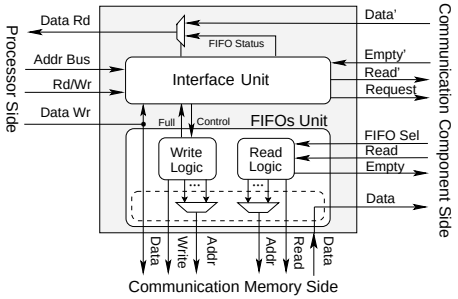


Figure 3: Communication Controller.

tion Memory (CM), it first checks if there is room in the corresponding FIFO by reading its status. If the FIFO is full, the processor blocks. Otherwise, it sends the data to the CC. The Interface Unit decodes the FIFO address sent by the processor along with the data and generates control signals (select FIFO, write data, or read status) to the write logic of the FIFOs Unit. The latter implements the multi-FIFO behavior. For each FIFO buffer the FIFOs Unit contains read and write counters that indicate the read and write positions into the buffer. These counters are used as read/write address generators and their values are used for determining the empty/full status of a FIFO. The FIFOs Unit also includes a memory interface logic that realizes the access to the Communication Memory (CM) connected to the CC (bottom part of Figure 3). Notice that since we use dual-port memories and read and write logic are separated, a FIFO in a CM can be accessed for read and write operations simultaneously by different processors or two FIFOs in a CM can be accessed at a time – one for read operation and one for write operation.

Recall that a processor can access FIFOs located in other processors' CMs via a communication component for read operations only. First, the processor checks if there is any data in the FIFO 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 granted only if a communication line is available and there is data in the FIFO. If 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 Figure 3) 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. When data is read from a FIFO of a CM, the signals to the read logic of the FIFOs Unit (*FIFO Sel* and *Read*) are generated by the communication component (the bottom part of the *communication component side* in Figure 3) as a response to a request from another CC.

The described blocking mechanism for accessing the CMs has to be done in the processors. The blocking can be realized in hardware (usually processors have dedicated embedded hardware to stall the processor) or in software by executing empty loops. We use the latter approach because it is more general. Different processors are stalled in hardware in different ways and therefore our CC would have to be aware of many possibilities. This would result in a more complex and less generic controller. Realizing the blocking mechanism in software makes the controller more generic, thereby simplifying the integration of different types of processors into a multiprocessor system.

3.4 Crossbar Communication Component

In this subsection we present the implementation of our Crossbar communication component. Our general approach to connect processors that communicate data through communication memories (CM) with FIFO organization allows the crossbar structure to be very simple. This results in a smaller crossbar with a reduced number of communication and routing resources and thus reducing the design area and power consumption. The structure of our crossbar component consists of two main parts, crossbar switch (CBS) and crossbar controller (CBC). The CBS implements uni-directional connections between communication memories and processors – recall that a processor uses a communication component only to read data. Due to the uni-directional communications and the FIFO organization of CMs, the number of signals and busses that has to be switched by our crossbar is reduced a lot. Since the addresses for accessing CMs are generated locally by the CCs, address busses are not switched through the crossbar. The crossbar switches 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.

The requests for granting a connection generated by the CCs are processed by the crossbar controller (CBC) using Round-Robin policy. If a request is for granting a connection, the CBC checks its request table whether the required connection is available at the moment. The request table contains information about the status (available or not available) of all connections. The table is updated each time a connection is granted or released.

3.5 Point-to-Point Network

In this section we describe how we implement a point-to-point communication in our platforms. In point-to-point 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 crossbar 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

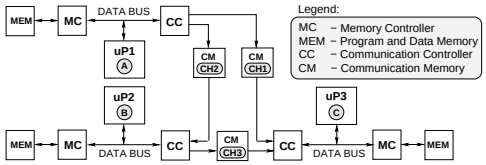


Figure 4: Point-to-Point Architecture.

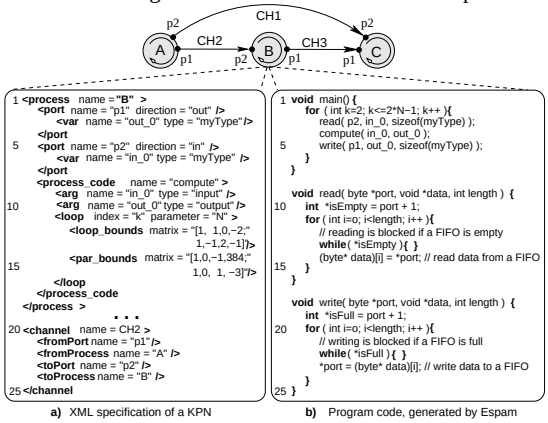
that each communication memory (CM) contains only one channel and each processor writes data only to its local CM (in compliance with our concept), our ESPAM tool synthesizes a point-to-point network in the following automated way. First, for each process in the KPN, ESPAM instantiates a processor together with a communication controller (CC). Then, ESPAM finds all the channels which the process writes to. For each found channel the tool instantiates a CM and assigns the channel to this CM. Finally, ESPAM connects the memory to the already instantiated processor. In Figure 4 we give an example of a point-to-point multiprocessor platform generated by ESPAM. Assume that the multiprocessor platform has to implement the KPN depicted in the top of Figure 5 and each process is executed on a separate processor. There are three channels that have to be assigned to three CMs. Following the procedure above ESPAM finds that CH1 and CH2 are written by process A – see the top part of Figure 5. Process A is assigned to be executed onto processor uP1 therefore CMs corresponding to CH1 and CH2 are instantiated and connected to uP1. Similarly, a CM corresponding to CH3 is instantiated and connected to processor uP2. Process C is assigned to processor uP3 and since process C only reads data from CH1 and CH3 no more CMs are instantiated. Processor uP3 is simply connected to the already instantiated CMs corresponding to CH1 and CH3. Notice that in Figure 4, a CC is connected to more than one CM. As we mentioned in Section 3.2, if a CM contains only one FIFO a CM is implemented by a dedicated FIFO component. Therefore, to connect one or more FIFOs to a processor in the case of point-to-point network, we use a very simplified version of our communication controller (CC) described in

Section 3.3. The simplified CC only translates the processor data bus signals to FIFO input/output signals. The CC is parameterized and it supports up to 128 FIFOs for read and write operations.

4. AUTOMATED PROGRAMMING

Application Specification

The first step to program multiprocessor systems in our ESPAM design methodology is the partitioning of an application into concurrent tasks where the inter-task communication and synchronization is *explicitly* specified in each task. The partitioning of an application into concurrent tasks can be done by hand or automatically [2, 10] and it allows each task or group of tasks to be compiled separately by a standard compiler in order to generate an executable code for each processor in the platform. The result of the partitioning done by the tools is an XML description of a Kahn Process Network (KPN) as an Approximated Dependence Graph (ADG) data structure [11]. It is a compact mathematical representation of the process network in terms of polyhedra. This allows formal operations to be defined and applied [11] on the KPN in order to generate an efficient code for the processors. A



a) XML specification of a KPN **b)** Program code, generated by Espam

Figure 5: Kahn Process Network Example.

simple example of a KPN is shown in Figure 5. Three processes (A, B, and C) are connected through three FIFO channels (CH1, CH2, and CH3). For the sake of clarity, in Figure 5a, we show the XML description only for one process (B). Process B has one input port and one output port defined in lines 2-7. In our example, process B executes a function called $compute$ (line 8). The function has one input argument (line 9) and one output argument (line 10). The relation between the function arguments and the ports of the process is given in lines 3 and 6. The function has to be executed $2 * N - 2$ times as specified by the polytope in lines 12-13. The value of N is between 3 and 384 (lines 14-15). Lines 20-25 show an example of how the topology of a KPN is specified: CH2 connects processes A and B through ports p1 and p2.

Code Generation

ESPAM takes the XML specification of an application, applies some operations [11] on it and automatically generates software (C/C++) code for each processor. The code contains the main behavior of a process, together with the blocking read/write synchronization primitives and the memory map of the system. The C code generated by ESPAM for process B is shown in Figure 5b. In accordance with the XML application specification, for loop is generated in the $main$ function of process B (lines 2-6) to execute $2 * N - 2$ times function $compute$. The C/C++ code implementing function $compute$ has to be provided by the designer. The function uses local variables in_0 and out_0. For simplicity, the declaration of the local variables is not shown in the figure. ESPAM inserts a read primitive to read from CH2, initializing variable in_0 and a write primitive to send the results (the value of variable out_0) to CH3 (Figure 5b, lines 3 and 5). The code of the synchronization read/write primitives, shown in the same figure, is automatically generated by ESPAM as well. Each primitive has 3 parameters. Parameter $port$ is the address of the memory location through which a processor can access a given FIFO channel. Parameter $data$ is a pointer to a local variable and $length$ specifies the amount of data (in bytes) to be moved from/to the local variable to/from 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 10 and 19. The blocking is implemented by $while$ loops with empty bodies in lines 13 and 22. Each empty loop iterates (does nothing) while a channel is full or empty. Then, in lines 14 and 23 the actual data transfer is done.

5. EXPERIMENTS AND RESULTS

In this section we present some of the results we have obtained by implementing and executing a Motion JPEG (M-JPEG) encoder application onto several multiprocessor platform instances using our ESPAM system design flow presented in Section 2. The main objective of this experiment is to show that our design flow successfully closes the implementation gap between the System and RTL abstraction levels of description as well as to show that using the ESPAM tool a very accurate exploration of the performance of alternative multiprocessor platforms based on real implementations becomes feasible since the design time is reduced significantly. For the implementations we used a prototyping board with one Xilinx FPGA.

Design Time

In Table 1 we show the processing times of each step in the design flow for the implementation of one platform instance. As described in Section 2, the inputs to our system design flow are the *Application*, *Platform*, and *Mapping Specifications*. The *Application Specification* has to represent the M-JPEG application as a KPN. For a certain class of applications the generation of KPNs is automated by the *translator* tools presented in [2, 10]. We started with the M-JPEG application given as a sequential C program. With small modifications we structured the C code in order to comply with the input requirements of the translators. Then we derived a KPN specification automatically. It took us about half an hour to modify the C code and just 22 seconds to derive the KPN specification. Notice that this is a one-time effort only

Table 1: Processing Times (hh:mm:ss).

	KPN Derivation	System Level to RTL conversion	Physical Implement.	Manual Modific.
Translators	00:00:22	–	–	00:30:00
ESPAM tool	–	00:00:24	–	00:10:00
XPS tool	–	–	02:09:00	–

because in the implementation of each new platform the same KPN specification is used. For each platform we wrote the *Platform* and *Mapping Specifications* by hand in approximately 10 minutes. This is a very simple task because our specifications are at a high system level of abstraction (not RTL level). Having all three system level specifications, our ESPAM tool converts them to RTL level specifications within half a minute. The generated specifications are close to an implementation and are automatically imported to the Xilinx Platform Studio (XPS) tool for physical implementation, i.e., mapping, place, and route onto our prototyping FPGA. Table 1 shows that it took the XPS tool more than 2 hours for the physical implementation. The reported time is for a platform instance containing 8 $MicroBlaze$ processors. However, in case of 2 processors XPS needs only 20 minutes. All tools run on a Pentium IV machine at 1.8GHz with 1GB of RAM.

The figures in Table 1 clearly show that a complete implementation of a multiprocessor system starting from high abstraction system level specifications can be obtained in about 2 hours using our ESPAM tool together with the translators [2, 10] and the commercial XPS tool. So, a significant reduction of design time is achieved. This allows us to explore the performance of 16 platforms using real system implementations. We implemented and ran the M-JPEG application on 16 alternative platforms using different *Mapping* and *Platform Specifications* in a very short amount of time, approximately 2 days.

Performance Results

The platforms contain up to 8 $MicroBlaze$ processors connected either point-to-point (P2P) or connected through a crossbar (CB). We compared the output data of each implementation with reference data in order to check that the implemented behavior is correct. For each multiprocessor system we measured the exact number of clock cycles needed to process an image of size 128 by 128 pixels. These numbers, depicted in the left part of Figure 6, are taken from simple hardwired timers and counters automatically integrated by ESPAM in each platform. The numbers indicate that for the M-JPEG application and the

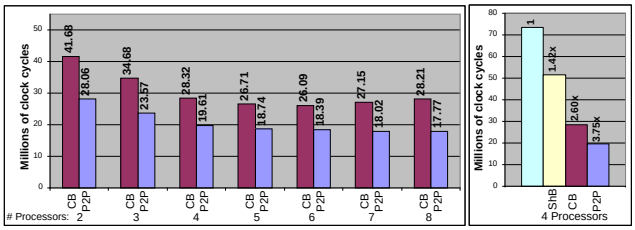


Figure 6: Performance Results.

alternative platforms we have experimented with, it is not reasonable to go beyond 4 $MicroBlaze$ processors. The performance speedup of systems with 4 $MicroBlase$s is depicted in the right part of the same figure. The left most bar shows the performance of the M-JPEG application run on one $MicroBlaze$. We use this as a reference point. We achieved performance speedup of 2.60x for the system with 4 processors and a crossbar component (CB) and for the system with 4 processors and point-to-point (P2P) connections the performance speedup is 3.75 whereas the theoretical maximum is 4x. For comparison we implemented also a multiprocessor platform with 4 processors connected through a shared bus (ShB) with Round-Robin arbitration policy. The achieved speedup by this multiprocessor system (the second bar) is only 1.42x. This clearly shows that a shared bus architecture is not an efficient architecture for building high-performance multiprocessor systems. By experimenting with up to 8 processors and with different platform topologies, we show that using our system design methodology and ESPAM tool it is a matter of hours to explore the performance of alternative multiprocessor systems by real implementations and measurements of actual numbers. These numbers are a 100% accurate. Gathering these numbers is faster than running cycle accurate simulations of the generated platforms. We do not know how much time is needed for an experienced designer to verify a simulation at RTL level 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 hours. Of course, performing simulation at a higher level of abstraction is even faster but the 100% accuracy of the numbers cannot be achieved.

Synthesis Results

In Table 2 we present the overall resource utilization of the multiprocessor systems with 4 processors we considered in our experiments. We also present the utilization results for the communication controllers (CC), a 4-port crossbar component (CB), and a 4-port shared bus component (BUS). The FPGA resources are grouped into slices that contain 4-Input Look-Up 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. Also, the last three rows show that our communication component (CB or BUS) together with the CCs in each system utilize a very small portion of the FPGA slices – around 5%. These numbers clearly indicate that our approach to connect processors through communication components and communication memories is very efficient in terms of slice utilization. The last column in Table 2 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 communication memories because for each M-JPEG system we use a maximum of 9 BRAM blocks

Table 2: 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%)	—

to implement FIFO buffers, distributed over 4 communication memories. 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 4 microprocessors in our platforms.

6. CONCLUSIONS

In this paper we presented our system design methods and techniques implemented in the ESPAM tool for automated platform synthesis, implementation, and programming. This automation significantly reduces the design time starting from system level specification and going down to complete implementation, thereby successfully closing the *Implementation Gap* described in Section 1.1.

Based on our experience with a Motion-JPEG encoder application, we conclude that our ESPAM tool, together with the translators [2, 10] and the XPS tool, is able systematically and automatically to implement and to program a multiprocessor platform very fast (within 2 hours). Thus, a very accurate exploration of the performance of alternative platforms at implementation level is now feasible in a relatively short amount of time. We implemented the Motion-JPEG application onto 16 alternative multiprocessor platforms in approximately 2 days.

The results presented in this paper show that our approach of connecting processors through communication controllers and communication memories is efficient in terms of HW utilization and performance speedup. For an M-JPEG encoder application implemented with 4 processors the communication logic utilizes only 5% of the resources. We achieved a speedup close to the theoretical maximum (4x) as compared to a single processor system.

7. REFERENCES

[1] Gilles Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. of the IFIP Congress 74*. 1974, North-Holland Publishing Co.

[2] Todor Stefanov et al., "System Design using Kahn Process Networks: The Compaan/Laura Approach," in *Proc. Int. Conference Design, Automation and Test in Europe (DATE'04)*, Paris, France, Feb. 16-20 2004.

[3] M.J. Rutten et all., "A Heterogeneous Multiprocessor Architecture for Flexible Media Processing," *IEEE Design & Test of Computers*, vol. 19, no. 4, 2002.

[4] Andy Pimentel et al., "A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels," *IEEE Transactions on Computers*, vol. 55, no. 2, 2006.

[5] Erwin de Kock, "Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study," in *Proc. 15th Int. Symposium on System Synthesis*, Japan, 2002.

[6] Kees Goossens et al., "Guaranteeing the Quality Of Services in Networks On Chip," in *Networks on Chip*. 2003, Kluwer Academic Publishers.

[7] B. Dwivedi et al., "Automatic Synthesis of System on Chip Multiprocessor Architectures for Process networks," in *Proc. Int. Conference on Hardware/Software Codesign and System Synthesis*, Sweden, 2004.

[8] D. Lyonnard et al., "Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip," in *Proc. 38th Design Automation Conference (DAC'2001)*, Las Vegas, USA, June 18-22 2001.

[9] A. Gerstlauer and D. Gajski, "System-level abstraction semantics," in *Proc. 15th Int. Symposium on System Synthesis (ISSS'02)*, Kyoto, Japan, Oct. 2-4 2002, pp. 231–236.

[10] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov, "Improved Derivation of Process Networks," in *4th Workshop on Optimization for DSP and Embedded Systems, ODES-4*, New York, USA, Mar. 2006.

[11] Todor Stefanov and Ed Deprettere, "Deriving Process Networks from Weakly Dynamic Applications in System-Level Design," in *Proc. 1th Int. Conf. on Hardware/Software Codesign and System Synthesis*, Newport Beach, California, USA, Oct. 1-3 2003, pp. 90–96.