Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 726096, 15 pages
doi:10.1155/2008/726096

Research Article

Automated Integration of Dedicated Hardwired IP Cores in Heterogeneous MPSoCs Designed with ESPAM

Hristo Nikolov, Todor Stefanov, and Ed Deprettere

Leiden Embedded Research Center, Leiden Institute of Advanced Computer Science, Leiden University, Niels Bohrweg 1, 2333 CA, Leiden, The Netherlands

Correspondence should be addressed to Hristo Nikolov, nikolov@liacs.nl

Received 27 September 2007; Revised 20 February 2008; Accepted 7 April 2008

Recommended by Sandeep Shukla

This paper presents a methodology and techniques for automated integration of dedicated hardwired (HW) IP cores into heterogeneous multiprocessor systems. We propose an IP core integration approach based on an HW module generation that consists of a wrapper around a predefined IP core. This approach has been implemented in a tool called ESPAM for automated multiprocessor system design, programming, and implementation. In order to keep high performance of the integrated IP cores, the structure of the IP core wrapper is devised in a way that adequately represents and efficiently implements the main characteristics of the formal model of computation, namely, Kahn process networks, we use as an underlying programming model in ESPAM. We present details about the structure of the HW module, the supported types of IP cores, and the minimum interfaces these IP cores have to provide in order to allow automated integration in heterogeneous multiprocessor systems generated by ESPAM. The ESPAM design flow, the multiprocessor platforms we consider, and the underlying programming (KPN) model are introduced as well. Furthermore, we present the efficiency of our approach by applying our methodology and ESPAM tool to automatically generate, implement, and program heterogeneous multiprocessor systems that integrate dedicated IP cores and execute real-life applications.

1. INTRODUCTION

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 (MPSoCs). There are several possibilities in building MPSoCs.

(1) *Homogeneous MPSoCs.* In these platforms, each processing component is a fixed ISA programmable processor, for example, IBM's *PowerPC* 405 [1], XILINX's *MicroBlaze* [2], and so on. The advantage of such systems is that in order to generate an executable code, a standard (e.g., C/C++) compiler is used. This leads to high flexibility in case different applications have to be mapped on a particular multiprocessor platform or a new version of an application is to be deployed on the same platform. However, the overhead in the executable code introduced by the compiler in general limits the performance these systems can achieve.

(2) *Multi-ASIP SoCs.* The advances in the *application specific instruction-set processors (ASIPs)* technology [3] raise an alternative to build high-performance systems. The ASIPs extend the fixed ISA processors to the so-called (re-)configurable or extensible processors. Examples of this approach are Xtensa LX configurable processor from (Tensilica, Santa Clara, Calif, USA) [4], IXP1200 network processor from (Intel, Santa Clara, Calif, USA) [5], and TriMedia TM3270 media processor from (NXP, Eindhoven, The Netherlands) [6]. The ASIPs approach enables system designers to add application- (domain) specific extensions (instructions) to a base processor that may have never been considered or imagined by designers of the original architecture. The addition of highly customized instructions matched perfectly to a specific application gives ASIPs the ability to deliver performance higher than the conventional fixed ISA

processors. As a consequence, building multi-ASIP systems would give better performance than MPSoCs utilizing fixed ISA processors. Although several design methodologies for building ASIPs exist [7, 8], building a multi-ASIP system is yet far from trivial and faces the same problems as any multiprocessor system, that is, synchronizing the interprocessor data communications and programming the system as a whole (more sophisticated parallelizing compilers are required). Moreover, to efficiently customize a multi-ASIP platform, additional design effort and tools are required.

(3) *Heterogeneous MPSoCs consisting of programmable processors and dedicated HW IP cores.* The *programmable* processors in such MPSoCs can be both fixed ISA and configurable (ASIP) processors. For efficiency, in a multiprocessor system different tasks have to be executed by different types of processing components which are optimized for execution of particular tasks. It is a common knowledge that higher performance is achieved by a dedicated (customized and optimized) HW IP core because it works more efficiently than fixed ISA and ASIP processors. Moreover, many companies already provide dedicated customizable IP cores optimized for a particular functionality that aim at saving design time and increasing overall system performance. Therefore, the idea of using dedicated IP cores in heterogeneous systems is very appealing. These heterogeneous systems are very attractive because they deliver high flexibility and high performance at the same time.

The ever increasing time-to-market pressure requires for systematic and, moreover, automated design methodologies for building MPSoCs where flexibility and IP reuse are very important aspects. However, two major problems emerge, namely, how to design, that is, how to integrate different dedicated HW IP cores and programmable processors into heterogeneous multiprocessor systems, and how to program these systems. The lack of standard interfaces that an IP core has to provide in order to allow seamless integration, and the lack of automated programming approaches for heterogeneous multiprocessor systems only exacerbates these problems.

1.1. Paper contributions

As a particular solution to the problems stated above, in this paper, we present our method and techniques for automated generation of *heterogeneous* multiprocessor systems where both fixed ISA processors and dedicated HW IP cores are used as processing components. Currently, ASIPs are out of the scope of this work because integrating ASIPs together with fixed ISA processors and HW IP cores adds an extra dimension in the complexity of the problems. Our approach is to solve the problems gradually, therefore, in this work, we investigated only the integration of third-party HW IP cores with fixed ISA processors. Our approach for integrating dedicated IP cores into a system is based on an automated HW module generation consisting of a wrapper around a predefined IP core. The structured, highly modularized, parameterized, and efficient design of our HW module is one of the contributions of this paper. This contribution substantially extends the embedded system-level platform synthesis and application mapping (ESPAM) tool [9] and allows automated generation and implementation of heterogeneous multiprocessor systems.

Another important contribution is that with ESPAM a heterogeneous multiprocessor system is programmed automatically where a part of the input application, initially specified as a sequential C program, is executed by programmable processors (ESPAM generates program code for each processor), and a part is implemented by dedicated IP cores. For the latter ESPAM integrates them by generating HW modules and connecting them to the other processing components of the system. Moreover, the structure of the HW module we propose allows an efficient integration and connection to alternative communication structures supported by ESPAM without the need of any additional modifications of the IP cores.

The success of our method and techniques for automated programming of the processors in a multiprocessor system and automated generation of HW modules for IP core integration into heterogeneous systems is based on the underlying programming (application) model used in ESPAM. ESPAM targets data-flow dominated (streaming) applications for which we use the Kahn process network (KPN) [10] model of computation. Many researchers [11–17] have already indicated that KPNs are suitable for efficient mapping onto multiprocessor platforms. However, decomposing an application into a set of concurrent tasks is one of the most time-consuming jobs imaginable [18]. Fortunately, our tool flow and programming approach combines a tool called PNgen [19] that we have developed for automatic derivation of KPN specifications from applications specified as sequential C programs.

The structure of the HW module we propose has been devised by carefully exploiting and efficiently implementing the simple communication and synchronization mechanisms of the KPN model. We have identified and developed a library, that is, a set of generic parameterized components used by ESPAM to compose an HW module. This is done by instantiating components from the library, connecting them, and setting their parameters in correspondence with the KPN specification. We consider our library of generic components as an important contribution because by making the HW module clearly structured and modularized, every component becomes more independent and loosely coupled. Therefore, we are able to design and optimize each component of the HW module separately. This brings much convenience for efficient and effective optimization, making the performance of the generated systems better.

Finally, we would like to mention that the ESPAM and PNgen tools are open source projects, available for third parties, and can be downloaded from [20].

1.2. 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 the paper.

1.2.1. Applications

One of the main assumptions for our work is that we consider only data-flow dominated applications in the realm of multimedia, imaging, and signal processing that naturally contain tasks communicating via streams of data. Such applications are very well modeled by using the parallel data-flow model of computation called Kahn process network (KPN). The KPN model we use is a network of concurrent autonomous processes that communicate data in a point-to-point fashion over bounded FIFO channels, using a blocking read/write on an empty/full FIFO as a synchronization mechanism. Furthermore, we consider KPNs that are input-output equivalent to static affine nested loop programs (SANLP). The properties of such programs are discussed in Section 1.2.5. We are interested in this subset of KPNs because they are analyzable at compile time (e.g., FIFO buffer sizes and execution schedules are decidable) and, as we show in this paper, HW synthesis from them is possible. Moreover, such KPNs can be derived automatically from the corresponding sequential programs [19, 21–23].

1.2.2. FIFO-based integration of IP cores

A key feature of our IP integration methodology is that the IPs are wrapped with a HW module and the communication is enabled through FIFOs. In order to achieve high performance with low overhead, our FIFO-based integration approach strictly follows the semantics of the KPN model of computation. Therefore, we motivate this approach by explaining the most favorable characteristics of the KPN model that we exploit in order to allow seamless and automated IP integration.

(1) The KPN model is determinate, which means that irrespective of the schedule chosen to run the network, the same input/output relation always exists. This gives a lot of scheduling freedom that can be exploited when integrating many different hardware IP cores in a heterogeneous multi-processor system.

(2) The interprocess synchronization in a KPN is done by a blocking read/write on empty/full FIFOs. This is a very simple synchronization protocol that can be realized easily and efficiently in hardware, thus making the synchronization, which is an essential part of our IP integration approach, very simple and efficient in terms of utilized hardware.

(3) The control in a KPN is completely distributed to the individual processes. Therefore, there is no global scheduler present. As a consequence, the integration of many hardware IP cores in a multiprocessor system is a simple task from the point of view that the local IP controllers do not have to be connected to and interact with a global scheduler controller.

(4) The exchange of data among processes in a KPN is distributed over the FIFOs. There is no notion of a single global data memory that has to be accessed by multiple processes. Therefore, when multiple IP cores are integrated into a multiprocessor system and communicate data via distributed FIFO buffers, resource contention is greatly reduced.

1.2.3. Multiprocessor platforms

We consider multiprocessor platforms in which the processing components, that is, programmable processors and/or HW IP cores, communicate data only through distributed memory units. Each memory unit can be organized as one or several FIFOs. The data communication among the processing components is realized by blocking read and write synchronization mechanism. Such platforms match and support very well the KPN operational semantics, thereby achieving high performance when KPNs are executed on the platforms. Also, compliant with the operational semantics of a KPN, our platforms support blocking synchronization mechanism, allowing the processors to be self-scheduled, avoiding a global scheduler component. If the number of processing components in a platform is less than the number of processes of a KPN, then some of the programmable processors execute more than one process. ESPAM schedules these processes at compile time and generates program code for a given processor which code does not require/utilize an operating system. In our approach, a HW IP core implements the computation of a single-KPN process. Therefore, we do not support more than one KPN process to be implemented by a single-HW IP core. Additional requirements for the HW IP cores are discussed in Section 3.2. The programmable processors and the HW IP cores in our platforms can be connected by a crossbar switch, a point-to-point network, or a shared bus. Additional details are given in Section 2.2.

1.2.4. Research and tools

The KPN model of computation has been widely studied in our group for more than 7 years. This research resulted in techniques and tools implemented in the Compaan/Laura design flow [11] for automated translation of SANLPs written in Matlab [21, 23] to KPN specifications targeting dedicated HW implementations on FPGAs [24]. Although these techniques are very advanced, they currently generate KPNs with too many FIFO channels which may lead to inefficient implementations. Also, they do not address the problem of what the FIFO buffer sizes should be. This is a very important problem because if the FIFO buffers are undersized, this leads to a deadlock in the KPN behavior.

We addressed the problems above, and based on the knowledge we have obtained working on Compaan, recently we have developed techniques for *improved* derivation of KPNs implemented in the PNgen tool [19]. These techniques allow for automated computation of efficient buffer sizes of the communication FIFO channels that guarantee deadlock-free execution of our KPNs. In addition, our group started a research on design automation for high-performance multi-processor systems, a challenging and very appealing domain nowadays. We devised a novel approach for automated *homogeneous* multiprocessor systems design, programming, and implementation. The concept and the techniques behind it were implemented in a tool called ESPAM [9]. Until recently, our approach was limited in the sense that only *homogeneous* systems containing programmable processors could be designed with ESPAM. Consequently, based on our experience with Laura [24], we developed techniques that substantially extend the functionality and flexibility of ESPAM allowing automated integration of programmable processors and dedicated IP cores into heterogeneous systems. As we mentioned, the IP core integration is based on generation of a HW module consisting of a wrapper around the IP. This approach originates from the general idea implemented in Laura, that is, generating of HW modules based on the properties of the KPN model we use. Although using the same concept in ESPAM, we developed different techniques in order to enable automated IP cores integration and connection to the other components of the system, that is, programmable processors and different communication structures. By carefully exploiting the main features of a KPN process, we created a library of parameterized components used to compose a HW module. In addition, we defined clear interfaces of the HW module and its components. This helped us to devise an efficient mechanism for connecting and synchronizing the components within an HW module keeping high performance of the integrated IP cores.

1.2.5. Tools inputs

Our ESPAM tool accepts as an input three specifications, that is, a platform specification, a mapping specification, and an application 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 platform instances 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 (only for programmable processors) of processes onto processing components. Based on this mapping, ESPAM determines automatically the most efficient mapping of FIFO channels onto distributed memory units. The application specification is a KPN in XML format derived from an application written as a static affine nested loop program (SANLP) in C (PNgen) or Matlab (Compaan). The SANLPs are programs that can be represented in the well-known polytope model [25]. That is, an 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 lower and upper bounds of the loops as well as all expressions in conditions and array accesses have to be affine functions of enclosing loop iterators and parameters. The parameters are symbolic constants, that is, their values may not change during the execution of the program. The above restrictions allow a compact mathematical representation of an SANLP, enabling the development of techniques for automated KPN derivation. Many applications in the domain we consider (see above) can be represented as SANLPs.

1.3. Related work

Several frameworks exist that address the issue of IP or component integration. Some of the most important are liberty simulation environment (LSE) [26], BALBOA [27], MCF [28], and Ptolemy [29]. In general, the methodologies proposed in these frameworks are related to our ESPAM methodology in the sense that we also propose a component (IP)-based approach to heterogeneous systems design. However, a major difference is that ESPAM addresses a different aspect of the heterogeneous systems design. ESPAM is a component-based framework targeting automated system synthesis, automated programming, and automated physical implementation of heterogenous multiprocessor systems whereas liberty, BALBOA, MCF, and Ptolemy are component-based frameworks/environments for system modeling and simulation.

The *control-read-execute-write* paradigm used to implement the operational semantics of our HW module wrapper is very similar to the paradigm used in [29, 30]. The difference is that in [29, 30], an implementation of similar paradigm is presented which is suitable and can be used only for hierarchical modeling and simulation purposes. In our ESPAM environment, we show an implementation of our *control-read-execute-write* paradigm that allows IP core integration in a physical multiprocessor system. Moreover, we present how our *control-read-execute-write* implementation is fully automatically synthesized/generated from sequential C code.

There are several approaches for HW design based on the ANSI C standard such as Handel-C (commercialized by Celoxica, Abingdon, Oxfordshire, UK [31]) and SpecC [32]. In contrast to our approach for multiprocessor systems design, Handel-C targets dedicated HW implementations on FPGAs. To express parallelism and event sensitivity in Handel-C, a designer has to use annotations (construct *par*) in the programming code. SpecC is a modeling language for the specification and design of embedded systems at system level. In [32], the authors propose a design methodology based on a library of reusable components that includes several steps which is similar to our methodology. The main difference, however, is that SpecC is an extension of the C programming language implying that the designer has to study it. Also, with SpecC the designer has to specify the possible parallelism of an application in an explicit way. In contrast to Handel-C and SpecC, the application specification in our methodology is a C program written by using a subset of the ANSI C standard without any special annotations, that is, SANLP explained in Section 1.2.5. A SystemC-based approach for design automation of digital signal processing systems is presented in [33]. The proposed methodology consists of an automated design space exploration, performance evaluation, and automatic platform-based system generation. Similarly to our approach, the input for the design flow contains an executable application specification (written in SystemC), a target architecture template (in both approaches built from components taken from a component library) and mapping constraints of the SystemC modules (in our methodology, we have a mapping giving a relation between the application and the architecture). In order to automate the design process, the SystemC application has to be written in a synthesizable subset of SystemC, called SysteMoC [34], whereas our restriction of the initial C program is to be an SANLP (see Section 1.2.5). The synthesizable subset of SystemC is required because for the IP core generation the authors use high-level synthesis tools which is a major difference with our concept for heterogeneous MPSoCs design. Instead, in this paper we propose an approach for dedicated IP core integration based on a HW module generation consisting of a wrapper around a predefined IP core.

In our automated design flow for MPSoC design, we use the KPN model of computation to represent an application and to map it onto alternative heterogeneous MPSoC architectures. A similar approach is presented in [35, 36]. Jerraya et al. propose a design flow concept that uses high-level parallel programming model to abstract HW/SW interfaces in the case of heterogeneous MPSoC design. The multiflex system presented in [37] is an application-to-platform mapping tool targeting multimedia and networking applications. Multiflex uses symmetric multiprocessing (SMP) and distributed system object component (DSOC) programming models. For implementation, the multiflex system targets the StepNP MPSoC platform architecture. Although the work presented in [35–37] targets heterogeneous MPSoC design, the authors do not address the problem of automated integration of dedicated HW IP cores into their heterogeneous MPSoCs. In contrast, in this paper we present efficient techniques for automated integration of IP cores into heterogeneous multiprocessor systems designed with ESPAM.

There are several initiatives such as VSIA [38] and OCP-IP [39] aiming at specifying "open" interface standards, for example, the virtual component interface (VCI) and the open core protocol (OCP), which will ease the integration effort required to incorporate IP cores into a system-on-chip. SPIRIT [40] is a consortium which aims at "enabling innovative IP reuse and design automation." IP-XACT is a standard defined by SPIRIT that allows to use general interfaces for connection between the IPs, where for each interface a reference bus definition is required. The main focus of these initiatives is to guarantee interoperability and reusability of a wide variety of IP cores in a "plug-and-play" fashion but this is achieved at the expense of more general, application-independent, and relatively slow interfaces and protocols. Our IP wrappers developed in ESPAM are not meant to be as general as VCI and OCP, that is, our wrappers support efficient integration of the specific type of IPs defined in Section 3.2. This fact and the fact that our wrappers are customized for every application, that is, they are automatically generated according to the KPN specification of an application, guarantee that the highest possible overall system performance is achieved.

The task transaction level (TTL) interface presented in [41] is a design technology for programming of embedded multiprocessor systems. Our programming approach is similar to TTL in the sense that both target streaming applications and both use communication primitives. However, in our approach, we consider only MPSoC architectures with distributed memory because such architectures give better timing performance compared to shared memory architectures. TTL is more flexible because it supports many communication primitives but programming an MPSoC by using TTL requires a lot of manual work which is hard (in some cases even impossible) to automate. In [10], Kahn proved that by using infinite FIFO queues, the blocking read in-order mechanism is sufficient to realize communication and synchronization in any streaming application modeled as a process network. Due to practical reasons, blocking write is needed as well because a FIFO implementation cannot have an infinite size. However, using a blocking write mechanism and finite memory resources may lead to deadlock of a KPN when executed. Therefore, we developed techniques for computing FIFO sizes such that a deadlock-free execution of our KPNs on our platforms is guaranteed [19]. In this sense, the blocking read and write, both in-order, form the minimum set of basic communication primitives realizing the communication mechanism of a process network when targeting real implementations. Other communication/synchronization mechanisms add more flexibility but at a certain price. In comparison with TTL, our platform model supports only the two basic primitives, which allows us to fully automate the programming of MPSoCs.

2. PRELIMINARIES

The paper is organized as follows. Here, in Section 2, we give an overview of our system design methodology and techniques centered around our ESPAM tool [9]. This is necessary in order to understand the main technical contribution of this paper presented in detail in Section 3, namely, automated IP core integration allowing automated heterogeneous MPSoCs generation, programming, and implementation with ESPAM. In Section 4, we present some results that we have obtained by using ESPAM to design efficient heterogeneous MPSoCs. Section 5 concludes the paper.

2.1. ESPAM design flow

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 written in XML format: (1) *Platform Specification* describing the topology of a multiprocessor platform; (2) *Application Specification* describing an application as a Kahn process network (KPN) [10], that is, network of concurrent processes communicating via FIFO channels. The KPN specification reveals the task-level parallelism available in the application. In the presented design flow, we start from a sequential program written in C and the PNgen tool [19] automatically converts it into a KPN specification. (3) *Mapping Specification* describing the relation between all processes and FIFO channels in *Application Specification* and all components in *Platform Specification*. The platform and the mapping specifications can be written by hand or can be generated automatically after performing a design space exploration. For this purpose, we use the Sesame tool [14]. The System-Level specification is given as input to ESPAM. First, ESPAM constructs a platform instance from the platform description. The platform instance is

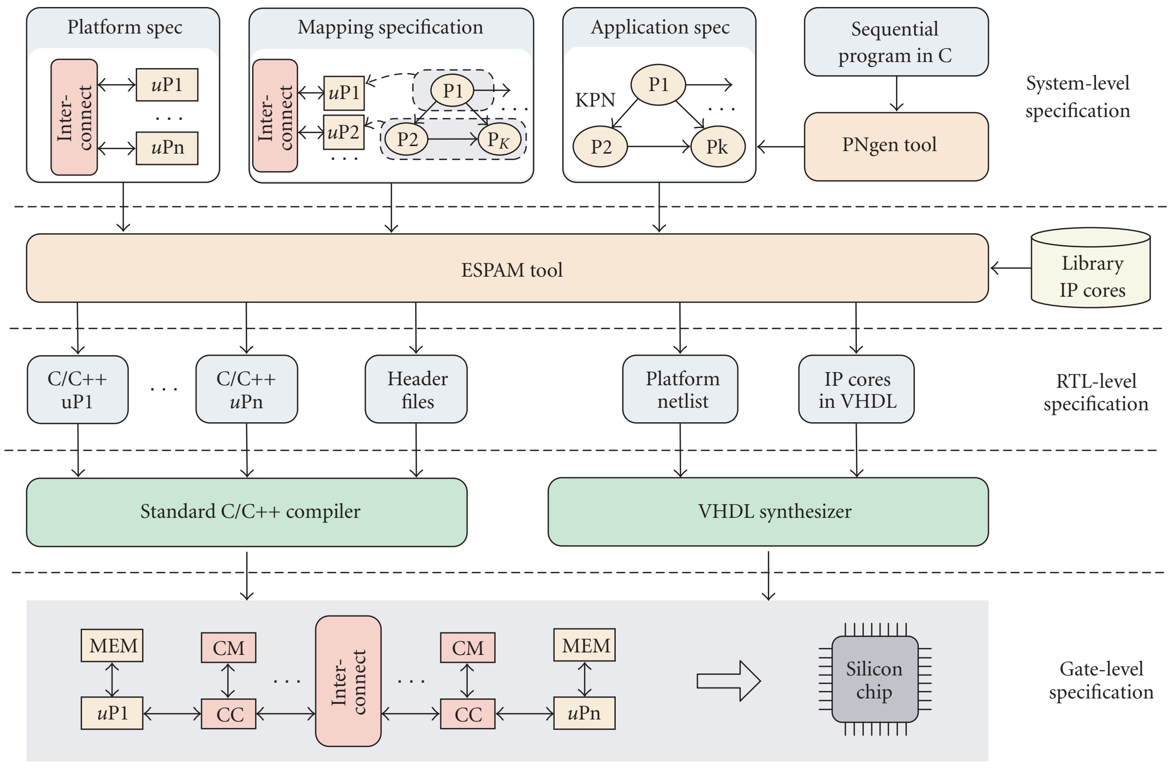


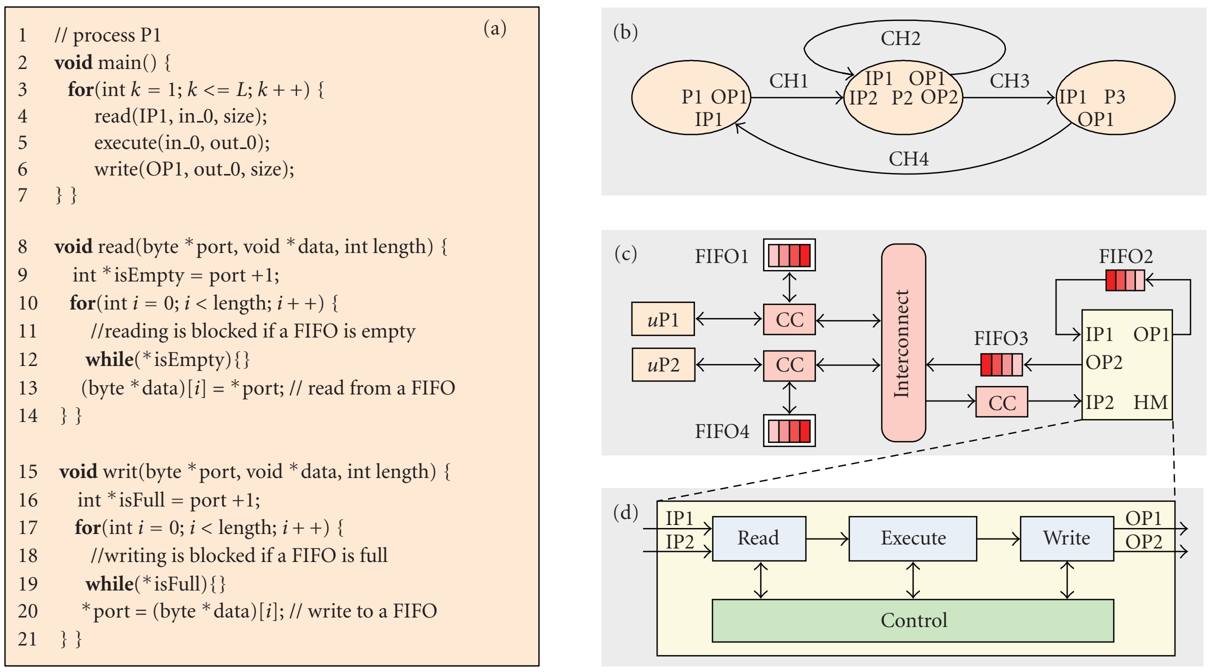
Figure 1: ESPAM system design flow.

an abstract model of an MPSoC because, at this stage, no information about the target physical platform is considered. The model defines only the key system components of the platform and their attributes. Second, ESPAM refines this abstract platform model to an elaborate parameterized RTL model (RTL-Level specification of an MPSoC) using the components from the IP library (see Figure 1). The refined system components are instantiated by setting their parameters based on the target physical platform features. Finally, ESPAM generates C/C++ program code for each processor in the multiprocessor platform in accordance with the application and mapping descriptions. The program code is further given to a standard GCC compiler to generate executable code. In addition, a commercial synthesizer converts the RTL-Level HW 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.

2.2. Multiprocessor platforms

Our ESPAM flow presented in [9] only supports automated design of *homogeneous* multiprocessor platforms, that is, platforms containing only programmable processors. Here, we briefly describe these platforms in order to show and describe clearly, later in Section 3, how we extend ESPAM to support automated design of *heterogeneous* multiprocessor platforms. The homogeneous multiprocessor platforms considered in [9] are constructed by connecting *processing*, *memory*, and *communication* components using *communication controllers (CCs)*. Our approach is explained below using the example of a multiprocessor platform depicted in the bottom-left part of Figure 1. It contains several processors connected to a communication component (INTERCONNECT) using communication memories (CMs), and communication controllers (CCs). The processors have separate program/data memory (MEM) and transfer data between each other through the CM memories. A communication controller connects a communication memory to the data bus of the processor it belongs to and to a communication component. Each CC implements the processor's local bus-based access protocol to the CM for write operations and the access to the communication component (INTERCONNECT) for read operations. 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. Each CM is organized as one or more FIFO buffers. We have chosen such organization because, then, the interprocessor synchronization in the platform can be implemented in a very simple and efficient way by blocking read/write operations on empty/full FIFObuffers located in the communication memory.

The FIFO organization of the CMs is implemented by the CCs in hardware which leads to very efficient data transfer, that is, a processor writes/reads a 32-bit word to/from a FIFO in 4 clock cycles [9]. If a processor supports direct FIFO communication, for example, the XILINX's *MicroBlaze* processor [2] used in ESPAM has several dedicated FIFO (FSL) interfaces (similar to Tensilica's Xtensa LX), then a CC implements a single FIFO which is directly connected to the



```
(a)
1   // process P1
2   void main() {
3     for(int k = 1; k <= L; k + +) {
4         read(IP1, in_0, size);
5         execute(in_0, out_0);
6         write(OP1, out_0, size);
7   } }

8   void read(byte *port, void *data, int length) {
9     int *isEmpty = port +1;
10    for(int i = 0; i < length; i + +) {
11        //reading is blocked if a FIFO is empty
12       while(*isEmpty){}
13      (byte *data)[i] = *port; // read from a FIFO
14  } }

15  void writ(byte *port, void *data, int length) {
16    int *isFull = port +1;
17    for(int i = 0; i < length; i + +) {
18        //writing is blocked if a FIFO is full
19       while(*isFull){}
20      *port = (byte *data)[i]; // write to a FIFO
21  } }
```

Figure 2: Example of heterogeneous MPSoC generated by ESPAM.

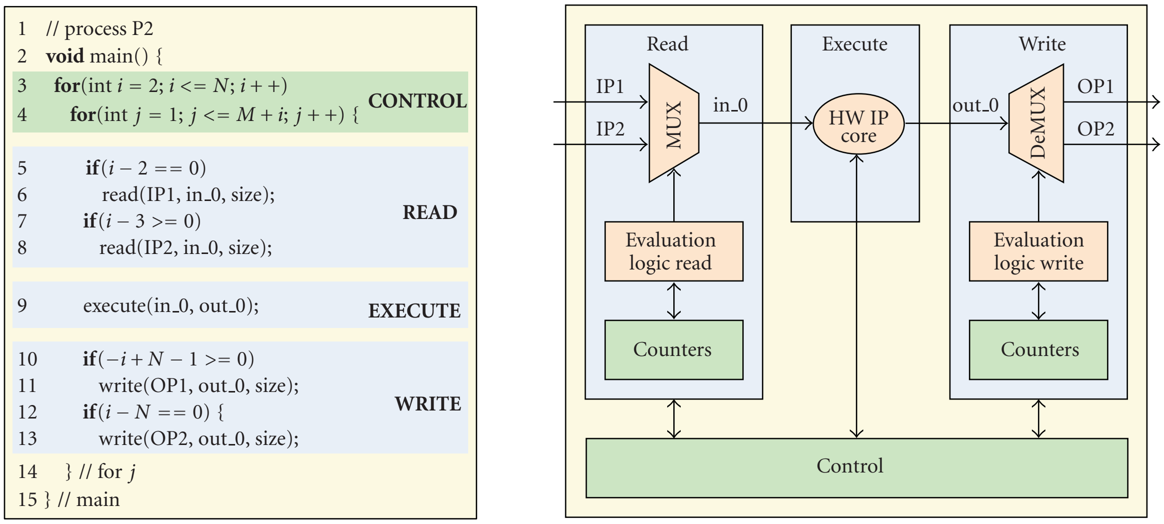
data path of the processor (through an FSL). In this case, the *MicroBlaze* processor uses a specific instruction to access the FIFO, reducing the transferring time to only 2 clock cycles (if blocking does not occur). The reduced time is caused by the fact that in the FSL communication, the blocking synchronization is supported and implemented by the state machine of the processor. For some applications, however, the number of FIFOs connected to a processor may exceed the number of direct FIFO links supported by the processor. In this case, ESPAM first utilizes all the available direct FIFO interfaces and then connects the remaining FIFOs to the processor data bus.

2.3. *Automated programming*

Our methodology and tool-flow for multiprocessor system design allow automated synthesis, programming, and implementation of multiprocessor platforms. Automated programming of an MPSoC means that the ESPAM tool automatically generates program code for each processor in the system, generates the memory map of the system, and generates code that implements the synchronization and communication between the processors. In our methodology and design flow, the first step is partitioning of an application into concurrent tasks where the intertask communication and synchronization is *explicitly* specified in each task. Such partitioning, performed automatically by the PNgen tool [19], allows each task or group of tasks to be compiled separately by a standard *C* compiler in order to generate an executable code for each processor in the platform. The partitioning of an application into concurrent tasks requires the use of a parallel model of computation in order to functionally specify the application. In our case, PNgen uses and generates such model, namely, a Kahn process network (KPN), for a given application initially specified as a sequential static affine nested loop C program. A KPN generated by PNgen is a network of concurrent autonomous processes that communicate data in a point-to-point fashion over FIFO channels, using blocking read/write on empty/full FIFOs as a synchronization mechanism. Moreover, regardless of the functional behavior specified by processes in a KPN generated by PNgen, always ESPAM takes each process specification and generates a specific code for each process where the structure of the code is the same for all processes to be executed by programmable processors. This uniform structure is explained by an example.

Consider the KPN shown in Figure 2(b). Three processes (P1, P2, and P3) are connected through four FIFO channels (CH1, CH2, CH3, and CH4). The structure of each process is the same and consists of a CONTROL part, a READ part, an EXECUTE part, and a WRITE part where the parts are specific—see, for example, process P2 in Figure 3(a). The same structure can be seen for process P1 in Figure 2(a)—see lines 2 to 7. The difference between P2 and P1 is in the specific code in each part. For example, the CONTROL part of P2 has two for-loops whereas the CONTROL part of P1 has only one for-loop. The READ part of P2 has two read primitives and if conditions specifying when to execute these primitives whereas the READ part of P1 has one read primitive executed unconditionally.

The blocking read/write synchronization mechanism of our KPNs is implemented by read/write synchronization primitives. They are the same for each process and are depicted in Figure 2(a)—see lines 8 to 21. Detailed explanation of the primitives' code can be found in [9]. The primitives are automatically generated and inserted in the



```
// process P2
void main() {
  for(int i = 2; i <= N; i++)
    for(int j = 1; j <= M+i; j++) {

      if(i - 2 == 0)
        read(IP1, in_0, size);
      if(i - 3 >= 0)
        read(IP2, in_0, size);

      execute(in_0, out_0);

      if(-i + N - 1 >= 0)
        write(OP1, out_0, size);
      if(i - N == 0) {
        write(OP2, out_0, size);
    } // for j
} // main
```

(a) (b)

Figure 3: Example of a HW module and its blocks' structure.

program code by ESPAM in the places where a process has to read/write data from/to a FIFO channel. For example, process P1 in Figure 2(a) reads data from its input channel via port IP1 (line 4). If data is not available, then the process blocks on reading until data arrives. Then it performs a computation on the data (line 5), and writes the result to its output FIFO channel via port OP1 (line 6). If the FIFO is full, then the process blocks on writing until there is room available in the FIFO. Lines 4 to 6 are repeated several times.

3. IP CORE INTEGRATION WITH ESPAM

In the previous sections, we presented our methodology for multiprocessor system design implemented in ESPAM. It allows automated generation of homogeneous multiprocessor platforms, that is, the processing components are only programmable processors. However, in many cases, a homogeneous system may not meet the performance requirements of an application. As an alternative, better performance can be achieved by using systems where different types of processing components execute different tasks. In general, a dedicated IP core delivers better performance than a processor which executes a software program for the same function. This motivated us to extend the ESPAM tool to support automatic generation of *heterogeneous* multiprocessor systems where both programmable processors and dedicated HW IP cores are used as processing components.

In this section, we present in detail our approach and techniques for automated IP core integration with ESPAM. It is based on a HW module generation that consists of a wrapper around a dedicated IP core. The basic idea in our approach is presented in Section 3.1. It is followed by a discussion on the type of the IPs supported by ESPAM, and the minimum interfaces these IPs have to provide in order to allow automated integration.. Then in Section 3.3, we give details about the internal structure of the HW module and the implementation of the wrapper. In Section 3.4, we explain how a HW module is automatically generated by ESPAM based on the KPN representation of the input application.

3.1. HW module—basic idea and structure

As we explained earlier, in the multiprocessor platforms we consider, the processors execute code implementing KPN processes, and communicate data between each other through FIFO channels mapped onto communication memories. Using communication controllers, the processors can be connected either point-to-point or via a communication component. We follow a similar approach to connect a dedicated IP core to other IPs or programmable processors in our platforms. We illustrate our approach with the example depicted in Figure 2. We map the KPN in Figure 2(b) onto the heterogeneous platform shown in Figure 2(c). Assume that process P1 is executed by processor *u*P2, P3 is executed by *u*P2, and the functionality of process P2 is implemented as a dedicated (predefined) IP core. Based on this mapping and the KPN topology, ESPAM automatically maps FIFO channels to communication memories (CMs) following the rule that a processing component only writes to its local CM. For example, process P1 is mapped onto processing component *u*P1 and P1 writes to FIFO channel CH1. Therefore, CH1 is mapped onto the local CM of *u*P1—see FIFO1 in Figure 2(c). In order to connect a dedicated HW IP core to other processing components, ESPAM generates a HW module (HM) that contains the IP core and a wrapper around it. Such an HM is then connected to the system using communication controllers (CCs) and communication memories (CMs), that is, an HM writes directly to its own local FIFOs and uses CC to read data from FIFOs located in CMs of other processors. This is illustrated in Figure 2(c)—see module HM that realizes process P2.

As explained in Section 2.3, our KPNs are derived automatically and the processes in our KPNs have always the same structure. It reflects the KPN operational semantics, that is, read-execute-write using blocking read/write synchronization mechanism. Therefore, a HW module realizing a process of our KPN has identical structure, shown in Figure 2(d), consisting of READ, EXECUTE, and WRITE blocks. A CONTROL block is added to capture the process behavior, for example, the number of process firings, and to synchronize the operation of the other three blocks.

The EXECUTE block of a HW module (HM) is actually a dedicated HW IP core to be integrated. It is not generated by ESPAM but it is taken from a library. The other blocks READ, WRITE, and CONTROL constitute the wrapper around the IP core. The wrapper is generated fully automatically by ESPAM based on the specification of a process to be implemented by the given HM. Each of the blocks in an HM has a particular structure which we illustrate with the example in Figure 3. Figure 3(a) shows the specification of process P2 generated by ESPAM if P2 would be executed on a programmable processor. We use this code to show the relation with the structure of each block in the HW modules generated by ESPAM, shown in Figure 3(b), if P2 is realized by dedicated HW.

In Figure 3(a), the read part of the code is responsible for getting data from proper FIFO channels at each firing of process P2. This is done by the code lines 5–8 which behave like a multiplexer, that is, the internal variable in_0 is initialized with data taken either from port IP1 or IP2. Therefore, the read part of P2 corresponds to the multiplexer MUX in the READ block of the HW module in Figure 3(b). Selecting the proper channel at each firing is determined by the if conditions at lines 5 and 7. These conditions are realized by the EVALUATION LOGIC READ component of block READ. The output of this component controls the MUX component. To evaluate the if conditions at each firing, the iterators of the for loops at lines 3 and 4 are used. Therefore, these for loops are implemented by HW counters in the HW module—see the COUNTERS component in Figure 3(b).

The write part in Figure 3(a) is similar to the read part. The only difference is that the write part is responsible for writing the result to proper channels at each firing of P2. This is done in code lines 10–13. This behavior is implemented by the demultiplexer DeMUX component in Figure 3(b). DeMUX is controlled by the EVALUATION LOGIC WRITE component which implements the if conditions at lines 10 and 12. Again, to implement the for loops, ESPAM uses the COUNTERS component. Although the counters correspond to the control part of process P2, ESPAM implements them in both the READ and WRITE blocks, that is, it duplicates the for loops implementation in the HW module. This allows the operation of blocks READ, EXECUTE, and WRITE to overlap, that is, they can operate in pipeline which leads to better performance of the HW module.

The execute part in Figure 3(a) represents the main computation in P2 encapsulated in the function call at code line 9. The behavior inside the function call is realized by the dedicated HW IP core depicted in Figure 3(b). As explained above, this IP core is not generated by ESPAM, but it is provided by a designer or it is a predefined third party IP core. In the HW module, the output of component MUX is connected to the input of the IP core and the output of the IP core to the input of component DeMUX. In our example, the IP core has one input and one output. In general, the number of inputs/outputs can be arbitrary. Therefore, every IP core input is connected to one MUX and every IP core output is connected to one DeMUX.

Notice that the loop bounds at lines 3-4 in Figure 3(a) are parameterized. The CONTROL block in Figure 3(b) allows the parameter values to be set/modified from outside the HW module at run time or to be fixed at design time. Another function of block CONTROL is to synchronize the operation of the other blocks and to make them to work in pipeline. Also, CONTROL implements the blocking read/write synchronization mechanism. Details are given in Section 3.3.

3.2. IP core types and interfaces

In this section, we describe the type of the IP cores that fit in our HW module idea and structure discussed above. Also, we define the minimum data and control interfaces these IPs have to provide in order to allow automated integration in MPSoC platforms designed with ESPAM.

(1) In our HW module, an IP core implements the main computation of a KPN process which in the initial specification is represented by a function call. Therefore, an IP core has to behave like a function call as well. This means that for each input data, read by the HW module, the IP core is *executed* and produces output data after an arbitrary delay.

(2) In order to guarantee seamless integration within the dataflow of our heterogeneous systems, an IP core must have unidirectional data interfaces at the input and the output that do not require random access to read and write data from/to memory. Good examples of such IP cores are the *multimedia cores* at http://www.cast-inc.com/cores/.

(3) To synchronize an IP core with the other blocks in our HW module, the IP has to provide "Enable/Valid" control interface. The "Enable" signal is a control input to the IP core and is driven by the CONTROL block in the HW module to enable the operation of the IP core when input data is read from input FIFO channels. If input data is not available, or there is no room to store the output of the IP core to output FIFO channels, then "Enable" is used to suspend the operation of the IP core. The "Valid" signal is a control output signal from the IP and is monitored by block CONTROL in order to ensure that only valid data is written to output FIFO channels connected to the HW module.

3.3. IP core wrapper implementation

As explained in Section 3.1, a HW module generated by our ESPAM tool contains a dedicated HW IP core and a wrapper around it. In this section, we give details about the structure and implementation of the IP Wrapper. We have already explained and shown that the structure of our HW

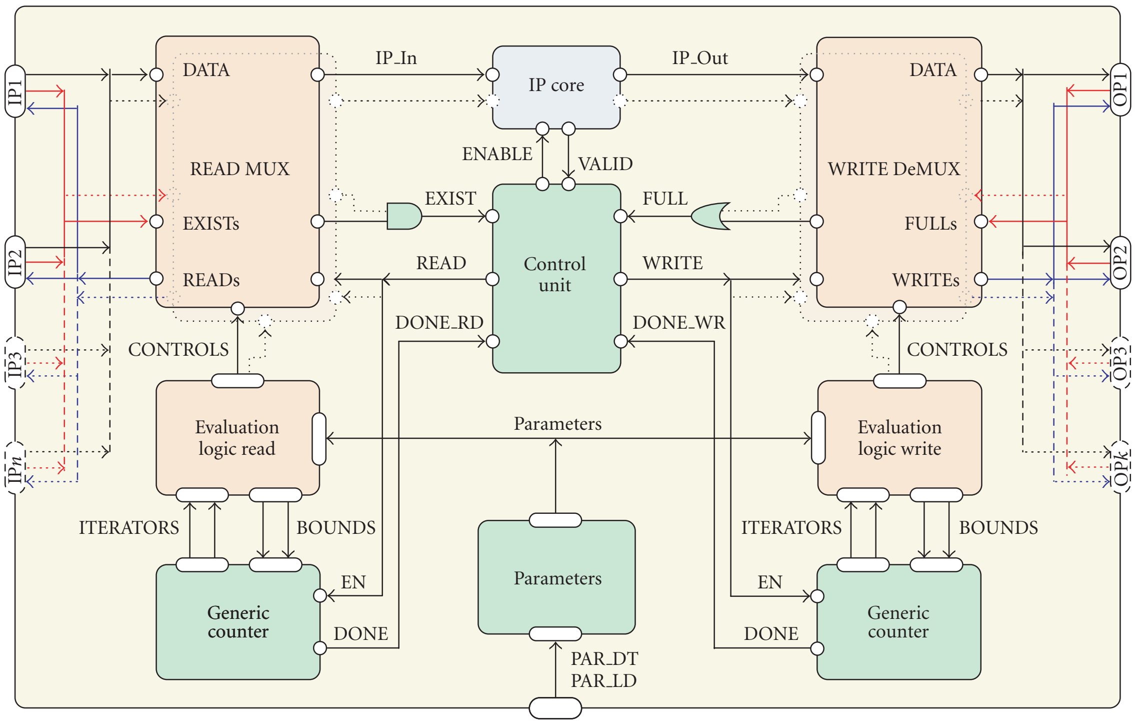


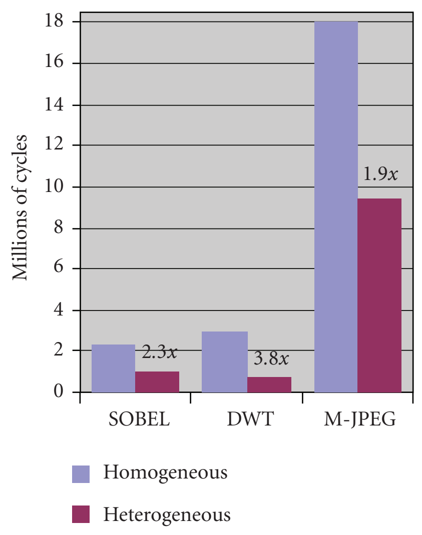
Figure 4: Detailed structure of the IP wrapper generated by ESPAM.

modules is the same regardless of the functional behavior they implement. This allows us to identify and implement the main building blocks discussed in Section 3.1 as a set of parameterized components with clearly defined interfaces, and by using these components to construct the IP Wrapper. Therefore, we have developed a HW module library of predefined parameterized components which ESPAM uses to generate HW modules. The generation is done by instantiating components from the library, connecting them, and setting their parameters according to the specification of processes in the input KPN. By making the IP wrapper (HW module, resp.) clearly structured and modularized, every component becomes more independent and loosely coupled. Therefore, we are able to develop and optimize each component separately which allows for efficient and effective optimization resulting in better performance of the generated systems. Below, we give more information about each component of the IP wrapper, its interface signals and its parameters.

The detailed structure of the IP wrapper is depicted in Figure 4. It contains several input and output data ports (IP1–IP*n* and OP1–OP*k*). The actual number is determined by the KPN topology. Each port implements a basic FIFO interface. It contains the minimum amount of signals that are present in any existing FIFO component. For the input ports, these are a DATA bus, EXIST, and READ signals. DATA is a bus used to read data from a FIFO connected to a port. EXIST is a signal indicating that a FIFO contains valid data to read and READ is a signal that controls the read enable of a FIFO. An output port implements a FIFO interface consisting of a DATA bus, FULL, and WRITE signals. DATA is a bus used to write data to an FIFO. FULL is a signal indicating when a FIFO is full and WRITE is a signal that controls the write enable of a FIFO. The IP wrapper has three global parameters defining its interfaces: number of input ports, number of output ports, and width of the data buses of the ports.

The data buses of the input interfaces are connected to the IP core inputs through multiplexers READ MUX. Recall that we use one READ MUX component for every input of an IP core. The READ MUX has two parameters: number of input ports (*#inputs*) and data width. Notice that each READ MUX port contains DATA bus, EXIST, and READ signals. Similarly, the outputs of the IP core are connected to the data buses of the output interfaces of the HW module using WRITE DeMUX components, one component per IP output. The parameters of component WRITE DeMUX are number of output ports (*#outputs*), each containing DATA bus, FULL, WRITE signals, and data width.

The CONTROL UNIT component, depicted in the center of Figure 4, synchronizes the operation of the IP wrapper components with the operation of the IP core. CONTROL UNIT does not have parameters. Therefore, its implementation and structure are exactly the same in any HW module generated by ESPAM. CONTROL UNIT generates READ and WRITE strobes to the FIFO channels and ENABLE signal to the IP core based on the EXIST and FULL signals coming from the FIFO channels. The DONE_RD and DONE_WR signals terminate the read/write actions. The implementation of CONTROL UNIT is very simple, that is, it implements only the three boolean equations described in Figure 6.



(a) Performance numbers

M-JPEG	Homogeneous		Heterogeneous	
VideoIn	11 K	4.1%	11 K	7.9%
DCT	135 K	50.8%	0.4 K	0.3%
Q	68 K	25.7%	68 K	50.3%
VLE	49 K	18.7%	49 K	39.5%
VideoOut	1.8 K	0.7%	1.8 K	2%

(b) M-JPEG tasks utilization

	SLICEs	FFs	LUTs
W1	221	190	371
W2	240	192	371
W3	208	147	361
W4	274	173	412
W5	269	173	390
W6	236	157	351

(c) HW resource utilization

Figure 5: Experimental performance and synthesis results.

READ = EXIST and not(FULL) and not(DONE_RD),

ENABLE = (not(DONE_RD) and READ), or
(DONE_RD and not(FULL) and not(DONE_WR))

WRITE = VALID and not(FULL) and not(DONE_WR).

Figure 6

The first equation shows the logic that enablesreading from FIFO channels. When data to be read is available, all the FIFOs where the result has to be stored are not full, and the read actions are not completed, signal READ is set to 1 and reading is initiated. The second equation is the logic that enables the operation of the IP core. It is enabled if the read actions are not completed and there is data to read. When the read actions are completed, the IP operation is enabled if the corresponding output FIFO channels are not full and the write actions are not completed. The third equation shows the logic that enables writing to FIFO channels. When the processed data is available at the output of the IP core indicated by the VALID signal, all the FIFOs where the data has to be stored are not full, and write actions are not completed, signal WRITE is set to 1, and writing is initiated.

As explained in Section 3.1, for loops are implemented as counters. Therefore, we have designed a parameterized GENERIC COUNTER component where only by setting parameters arbitrary nested for loops can be implemented. The parameters define the number of counters (*#counters*) and the *size* of each counter. GENERIC COUNTER has interface called BOUNDS that allows the lower and upper bounds of each counter to be set at run time. The GENERIC COUNTER components are controlled by enable signals (EN), that is, if a read or write action has been performed, the corresponding counter advances. When all the counters reach their upper bounds, this situation is indicated to CONTROL UNIT by signals DONE_RD and DONE_WR.

The parameters of the components discussed so far are so called *structural* parameters. They are set at design time and used to determine the components structure. However, processes in our KPNs may contain parameters that determine at run time the behavior of the processes. For example, the upper bounds of the for loops of process P2 in Figure 3(a) are parameterized by parameters N and M. We call such parameters *behavioral* parameters. These parameters are managed in our HW module by the PARAMETERS component. It provides an interface, PAR_DT, and PAR_LD, for loading behavioral parameter values from outside of the HW module at run time. PAR_DT is a data bus to transfer behavioral parameter values and PAR_LD is the write enable signal to store them in the PARAMETERS component. Like most of the components in our HW module, the PARAMETERS component itself has structural parameters that define the number of behavioral parameters (*#parameters*) and their default *values*.

The behavioral parameters in our KPNs are used to determine for-loop bounds and to evaluate if conditions that determine input/output ports selection as shown in Figure 3(a). In our IP wrapper, this is implemented in the two EVALUATION LOGIC components shown in Figure 4. The values of the behavioral parameters are propagated from the PARAMETERS component to these two components. Also, the values of the loop iterators implemented as counters in the GENERIC COUNTER components are propagated through the ITERATORS buses. In the two EVALUATION LOGIC components, the counter bounds are calculated and propagated back to the GENERIC COUNTER components at run time through the BOUNDS interface. Also, in the two EVALUATION LOGIC components the selection conditions are calculated to determine the values of signals CONTROLS.

These signals control components READ MUX and WRITE DeMUX that select from/to which FIFO channels data has to be read/written. The structural parameters of the EVALUATION LOGIC components are number of output control signals (*#control_signals*), number of counters (*#counters*), and number of behavioral parameters (*#parameters*). We would like to notice that the implementation of components EVALUATION LOGIC depends not only on the values of the structural parameters, but also on the specification of a KPN process realized by a HW module. Therefore, ESPAM synthesizes different EVALUATION LOGIC components for each HW module in our heterogeneous platforms. In contrast, the other components in any HW module are predefined and ESPAM just instantiates them from the HW module library and only assigns proper values to their structural parameters.

```
BOUND_LOW_i <= 2;
BOUND_HIGH_i <= PAR_N;
BOUND_LOW_j <= 1;
BOUND_HIGH_j <= PAR_M + ITERATOR_i
```

Figure 7

```
CONTROLS(0) <= (−iterator_i + par_N − 1) >= 0; - - OP1,
CONTROLS(1) <= (iterator_i − par_N) = 0; - - OP2
```

Figure 8

3.4. Automated generation of HW modules

The presented components of our HW module, their parameters and interfaces as well as the structure of the HW module allow the generation of the HW module to be automated in ESPAM. In this section, we explain how ESPAM automatically generates a HW module based on a KPN process specification. This is done in several steps, namely, components instantiation, parameters setting, and evaluation logic generation. For the sake of clarity, we illustrate these steps by an example of generating the HW module for process P2 in Figure 2(b). The process specification is given in Figure 3(a). The generated HW module is the module depicted in Figure 4 if the dashed components and connections are not considered.

First, ESPAM instantiates all the components of the IP wrapper and the IP core as follows. The IP wrapper requires two input (IP1, IP2) and two output (OP1, OP2) ports. This information is extracted from the KPN topology—see process P2 in Figure 2(b). The IP core that implements the main computation of process P2 in our example has one input and one output port. Therefore, ESPAM instantiates one READ MUX and one WRITE DeMUX components and connects them to the IP core and to the input/output ports of the HW module. Then ESPAM instantiates and connects the remaining components of the IP wrapper as shown in Figure 4. This is possible to be done automatically because the number of the remaining components and their connections do not change across different HW modules and it is predefined as a parameterized template in ESPAM. Second, ESPAM sets the values of the parameters of the instantiated components as follows:

READ MUX—*#inputs* = 2 because process P2 has two input ports;

WRITE DeMUX—*#outputs* = 2 because process P2 has two outputs;

PARAMETERS—*#parameters* = 2 because process P2 has two behavioral parameters N and M. The default values of N and M are set as well by extracting this information from the specification of process P2. This is possible because in our KPNs a behavioral parameter is defined by min, max, and default values;

GENERIC COUNTER—*#counters* = 2 because the body of process P2 contains two nested for-loops with iterators i and j. The parameter *size* of each counter is set according to the maximum value that each loop iterator can reach during operation. In our example, size = max(N) for the counter corresponding to loop iterator i. Similarly, size = max(M) + max(N) for the counter corresponding to loop iterator j;

CONTROL UNIT—no parameters are set because, as we explained in Section 3.3, this component is not parameterized;

EVALUATION LOGIC—*#control_signals* = 2 because of the two input and two output ports of the HW module, *#counters* = 2 because process P2 has two nested loops, and *#parameters* = 2 because P2 has two behavioral parameters N and M.

In our example, parameter *data_width* of all components is set to 32 bits.

Finally, as a last step of the automated generation of the HW module, ESPAM generates the implementations of the EVALUATION LOGIC components. The implementations contain two parts, that is, logic to calculate the lower and upper bounds of the counters and logic to calculate the values of the control signals propagated to the READ MUX and WRITE DeMUX components. According to the specification of process P2 at lines 3-4 in Figure 3(a), the counter bounds in both EVALUATION LOGIC components are implemented by logic synthesized from VHDL code generated by ESPAM as shown in Figure 7.

The read if conditions at code lines 5 and 7 and the write if conditions at code lines 10 and 12 in Figure 3(a) are also implemented by logic synthesized from VHDL code generated by ESPAM. In Figure 8, we show the code of the write if conditions in the EVALUATION LOGIC WRITE component.

4. EXPERIMENTS AND RESULTS

In this section, we present some of the results we have obtained by implementing and executing three applications, namely, a Sobel edge detection, a discrete wavelet transform

(DWT), and a motion JPEG (M-JPEG) encoder application, onto homogeneous and heterogeneous multiprocessor systems using our ESPAM tool and the design flow presented in Section 2.1. The main objective of this experiment is to show the effectiveness of our approach for HW IP core integration in terms of design time, achieved performance, and HW resource utilization of the generated HW modules. For prototyping purpose, we use an FPGA board with one Xilinx VirtexII-6000 device.

4.1. *Design time*

Following our design flow, we started with the three applications (Sobel, DWT, and M-JPEG) given as sequential C programs and automatically derived the *Application Specifications*, that is, KPNs using the PNgen tool in 5 minutes. Details about the derived KPNs are presented in [19]. For each application, we wrote the system-level *Platform* and *Mapping Specifications* by hand in XML format in 10 minutes. In this experiment, each of the three homogeneous platforms contains 5 *MicroBlaze* processors connected via crossbar network. Having all three input specifications for each application, our ESPAM tool generated and programmed a homogeneous multiprocessor system at RTL level, which was automatically imported to the Xilinx XPS tool for physical implementation onto our prototyping FPGA. The overall design and implementation time of each homogeneous system was about an hour.

We have performed similar actions as described above in order to generate three *heterogeneous* multiprocessor systems using our design flow. We had to modify only the initial system-level *Platform* and *Mapping Specifications* for each application in order to replace some of the *MicroBlaze* processors with dedicated HW IP cores. This took us less than 5 minutes. For the Sobel application, we used 3 *MicroBlaze* processors and 2 dedicated IP cores. The IP cores estimate the first derivative of an image intensity function. For the DWT application, we used 1 *MicroBlaze* processor and 4 dedicated IP cores. The IP cores are 2 low and 2 high-pass filters. For the M-JPEG application, we used 4 *MicroBlaze* processors and 1 discrete cosine transform (DCT) IP core. Again, the overall design and implementation time of each heterogeneous system was about an hour.

As explained above, in the heterogeneous systems we used several dedicated HW IP cores. They were written in synthesizable VHDL. For the Sobel and DWT applications, the IP cores have a simple structure, that is, they implement convolution-based operations. These IP cores have been developed by 2 master students and added to the ESPAM library in about one working day. For the M-JPEG application, we used an IP core that performs a discrete cosine transform (DCT) operation. We have downloaded this IP core from the Xilinx website at http://www.xilinx.com/bvdocs/appnotes/xapp_610.zip. In order to add this IP core to the ESPAM library, we had to make small modification related to the control (Enable/Valid) interface discussed in Section 3.2. The DCT IP core provided by Xilinx has "Valid" signal but it does not have "Enable" signal. We added this signal to the IP core and the IP core to the library within 30 minutes.

4.2. *Performance results*

The performance numbers we have obtained for the implemented multiprocessor systems are shown in Figure 5(a). For each multiprocessor system, we measured the exact number of clock cycles needed to process an image of size 128×128 pixels. As one may expect, the numbers in the figure show that the heterogeneous systems achieve better performance. This is because the dedicated HW IP cores, we use, work more efficiently than the *MicroBlaze* processors for the same functionality. What is more important to discuss here is the achieved speed-up, depicted in Figure 5(a) above the bars of the heterogeneous systems, in order to show the efficiency of our approach for HW module generation and IP core integration. Consider the performance results of the M-JPEG systems. The M-JPEG application consists of 5 tasks, namely, VideoIn, DCT, quantization (Q), variable length encoding (VLE), and video out. The left part of column HOMOGENEOUS in Figure 5(b) shows how many thousands of clock cycles it takes for a *MicroBlaze* processor to execute each task by processing one data token—an image block of size 8×8 pixels. The numbers in the next column show the same information in percentage of the overall processing time utilized by each task. It can be seen that the DCT is the bottleneck of the system taking more than 50% of the whole processing time for one block and consequently, for the whole image. These 50% mean that if the DCT is substituted with more efficient implementation, theoretically the overall performance of the system can be increased at most 2 times. The column HETEROGENEOUS in Figure 5(b) shows the clock cycles and the percentage of each task performed by the heterogeneous M-JPEG system where the DCT is implemented by a very fast dedicated HW IP core and integrated using our HW module generation approach. In this system, the DCT takes only 0.3% of the whole processing time. In this case, Figure 5(a) shows that the overall speed-up compared to the homogeneous system is 1.9x which is close to the theoretical maximum 2x for the heterogeneous system where only the DCT is a dedicated IP core. This clearly shows the efficiency of our approach for IP core integration by generating HW modules.

4.3. *Synthesis results*

Recall that a HW module generated by our tool ESPAM consists of an IP core and a wrapper around it where the IP core is given and only the wrapper is generated by ESPAM. Therefore, we present only the HW resource utilization of our generated wrappers in order to show how efficient our wrappers are in terms of utilized HW. In Figure 5(c), we present the resource utilization of the IP wrappers of six IP cores that we used in our experiments. Each row W1–W6 in Figure 5(c) corresponds to an IP wrapper. The utilized FPGA resources are grouped into slices that contain flip-flops and 4-input look up tables—see columns 2, 3, and 4, respectively. The numbers show low HW resource utilization which on average is 241 slices. Moreover, the number of resources utilized by a wrapper does not depend on the size of the IP core it integrates, that is, a larger IP core does not require a larger wrapper. For example, wrapper W3 of the DCT core utilizes only 208 slices, whereas the DCT IP core itself utilizes 1369 slices. Wrapper W2 of the IP core that estimates the first derivative in Sobel utilizes 240 slices, whereas the IP core itself utilizes 424 slices.

In general, the HW complexity of our wrappers is determined mainly by the number of MUX and DeMUX components, the number of counters implementing for loops of a KPN process, and the number of behavioral parameters of a KPN process. The three applications we used in our experiment process images. We specified the applications with two nested for loops that iterate through an image and we used two behavioral parameters as loop bounds, that is, image, width, and height. Since the number of for loops and behavioral parameters is the same for all wrappers in our experiment, the difference in the resource utilization of our wrappers is caused by the different input/output ports of the wrappers and the IP cores they integrate.

5. CONCLUSIONS

In this paper, we presented our method and techniques implemented in ESPAM for automated integration of dedicated hardwired IP cores into heterogeneous multiprocessor systems. The integration is based on a HW module generation that consists of predefined dedicated IP core and a wrapper around it. To allow automated IP core integration, our approach requires these IP cores to provide simple data and control interfaces. The proposed method for IP core integration was applied on several real-life applications. The reported results show that the integration is efficient in terms of performance and HW resource utilization.

ACKNOWLEDGMENT

This work was supported by the Dutch Technology Foundation (PROGRESS/STW) under the Artemisia project (LES.6389).

REFERENCES

[1] IBM PowerPC Wite Paper, http://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC 405 Embedded Cores.

[2] The Xilinx's Microblaze Processor, http://www.xilinx.com/products/design resources/proc central/microblaze arc.htm.

[3] K. Keutzer, S. Malik, and A. R. Newton, "From ASIC to ASIP: the next design discontinuity," in *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '02)*, pp. 84–90, Freiburg, Germany, September 2002.

[4] Tensilica, "Xtensa configurable processors," http://www.tensilica.com/products/xtensa-overview.htm.

[5] Intel Corp., "Intel IXP1200 Network Processor, Product Datasheet," December 2001.

[6] NXP, http://www.nxp.com/.

[7] M. Gries and K. Keutzer, *Building ASIPs: The Mescal Methodology*, Springer, New York, NY, USA, 2005.

[8] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications (Systems on Silicon)*, Morgan Kaufmann, San Francisco, Calif, USA, 2006.

[9] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and automated multiprocessor systen design, programming, and implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 3, pp. 542–555, 2008.

[10] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress on Information Processing*, pp. 471–475, North-Holland, Stockholm, Sweden, August 1974.

[11] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using Kahn process networks: the Compaan/Laura approach," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, vol. 1, pp. 340–345, Paris, France, February 2004.

[12] E. A. de Kock, G. Essink, W. J. M. Smits, et al., "YAPI: application modeling for signal processing systems," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 402–405, Los Angeles, Calif, USA, June 2000.

[13] A. Nieuwland, J. Kang, O. P. Gangwal, et al., *C-HEAP: A Heterogeneous Multi-processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems*, Kluwer Academic Publishers, Norwell, Mass, USA, 2002.

[14] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–112, 2006.

[15] E. A. de Kock, "Multiprocessor mapping of process networks: a JPEG decoding case study," in *Proceedings of the 15th International Symposium on System Synthesis (ISSS '02)*, pp. 68–73, Kyoto, Japan, October 2002.

[16] K. Goossens, J. Dielissen, J. van Meerbergen, et al., "Guaranteeing the quality of services in networks on chip," in *Networks on Chip*, pp. 61–82, Kluwer Academic Publishers, Hingham, Mass, USA, 2003.

[17] B. K. Dwivedi, A. Kumar, and M. Balakrishnan, "Automatic synthesis of system on chip multiprocessor architectures for process networks," in *Proceedings of the 2nd International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '04)*, pp. 60–65, Stockholm, Sweden, September 2004.

[18] G. Martin, "Overview of the MPSoC design challenge," in *Proceedings of the 43rd Design Automation Conference (DAC '06)*, pp. 274–279, San Francisco, Calif, USA, July 2006.

[19] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: a tool for improved derivation of process networks," *EURASIP Journal on Embedded Systems*, vol. 2007, Article ID 75947, 13 pages, 2007.

[20] "Espam and PNgen download link," http://daedalus.liacs.nl/Site/Download.html.

[21] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: deriving process networks from matlab for embedded signal processing architectures," in *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES '00)*, pp. 13–17, San Diego, Calif, USA, May 2000.

[22] T. Stefanov and E. Deprettere, "Deriving process networks from weakly dynamic applications in system-level design," in *Proceedings of the 1st International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '03)*, pp. 90–96, Newport Beach, Calif, USA, October 2003.

[23] A. Turjan, B. Kienhuis, and E. Deprettere, "Translating affine nested-loop programs to process networks," in *Proceedings of*

the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '04), pp. 220–229, Washington, DC, USA, September 2004.

[24] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere, "Laura: leiden architecture research and exploration tool," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL '03)*, pp. 911–920, Lisbon, Portugal, September 2003.

[25] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model*, vol. 1132 of *Lecture Notes in Computer Science*, pp. 79–103, Springer, London, UK, 1996.

[26] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August, "The liberty simulation environment: a deliberate approach to high-level system modeling," *ACM Transactions on Computer Systems*, vol. 24, no. 3, pp. 211–249, 2006.

[27] F. Doucet, S. Shukla, M. Otsuka, and R. Gupta, "BALBOA: a component-based design environment for system models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 12, pp. 1597–1612, 2003.

[28] D. A. Mathaikutty and S. K. Shukla, "MCF: a metamodeling based visual component composition framework," in *Advances in Design and Specification Languages for Embedded Systems*, pp. 319–337, Springer, New York, NY, USA, 2007.

[29] E. A. Lee, J. Liu, X. Liu, et al., "Ptolemy II: heterogeneous concurrent modeling and design in java," Tech. Rep. UCB/ERL M99/40, University of California, Berkeley, Calif, USA, 1999.

[30] H. D. Patel, S. K. Shukla, and R. A. Bergamaschi, "Heterogeneous behavioral hierarchy extensions for SystemC," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 4, pp. 765–780, 2007.

[31] Celoxica, http://www.celoxica.com/.

[32] J. Zhu, R. Doemer, and D. D. Gajski, "Syntax and semantics of the specC language," in *Proceedings of the 7th Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI '97)*, p. 8, Osaka, Japan, December 1997.

[33] C. Haubelt, J. Falk, J. Keinert, et al., "A systemC-based design methodology for digital signal processing systems," *EURASIP Journal on Embedded Systems*, vol. 2007, Article ID 47580, 22 pages, 2007.

[34] J. Falk, C. Haubelt, and J. Teich, "Efficient representation and simulation of model-based designs in systemC," in *Proceedings of the International Forum on Specification & Design Languages (FDL '06)*, pp. 129–134, Darmstadt, Germany, September 2006.

[35] A. A. Jerraya, A. Bouchhima, and F. Pétrot, "Programming models and HW-SW interfaces abstraction for multiprocessor SoC," in *Proceedings of the 43rd Design Automation Conference (DAC '06)*, pp. 280–285, San Francisco, Calif, USA, July 2006.

[36] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya, "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip," in *Proceedings of the 38th Design Automation Conference (DAC '01)*, pp. 518–523, Las Vegas, Nev, USA, June 2001.

[37] P. G. Paulin, C. Pilkington, M. Langevin, et al., "Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 7, pp. 667–679, 2006.

[38] VSIA, http://www.vsi.org/.

[39] OCP, http://www.ocpip.org/.

[40] SPIRIT, www.spiritconsortium.org/.

[41] 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 *Proceedings of the 2nd International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '04)*, pp. 206–217, Stockholm, Sweden, September 2004.