Transformations for polyhedral process networks

Meijer, S.

Chapter 6

Executing PPNs on Fixed Programmable MPSoC Platforms

In Chapter 1, we have indicated that the Daedalus tool-flow instantiates a specific hardware platform, called ESPAM [61] , prototyped on a FPGA to execute PPNs as efficiently as possible. Recall that we also argued in Chapter 1 that such a specific hardware platform may not always be available to a designer. Therefore, we want to investigate how to execute PPNs onto commercial-off-the-shelf (COTS), programmable MPSoC platforms. In this chapter, we address the issue how to execute polyhedral process networks onto COTS programmable MPSoC platforms and experiment with 2 different platforms: the Intel IXP network processor [1] and the CELL BE processor [39]. The Intel IXP is interesting as it has hardware support for FIFO communication to some extent, i.e., the IXP is highly optimized for streaming data, albeit in the form of internet packets. This makes the Intel IXP a dedicated streaming processing platform. As a second platform, we chose to experiment with a more general purpose MPSoC computing platform, i.e., the Cell platform, which lacks any hardware support for FIFO communication. For both platforms, there is a mismatch with the PPN model of computation. The mismatch is related to the FIFO read/write primitives used in the PPN model of computation and the way FIFO communication is supported by the hardware platform. This mismatch is the most evident in the Cell processor because it lacks any hardware support for FIFO communication, while the IXP has FIFO support to a certain extent. Taking this mismatch into account, we want to investigate in this chapter, how FIFO communication can be realized in the most efficient way using the provided communication infrastructure of these two COTS programmable MPSoC platforms.

6.1 The Programmable Platforms

In this section, we briefly discuss the Cell processor and the Intel IXP processor architectures, i.e., we discuss the interesting components of both platforms and explain the mismatch between the processor architectures and the PPN model of computation.

The Cell

The Cell BE platform [39] is a very good representative example of a state-of-the-art heterogeneous programmable MPSoC platform. A high-level schematic of the Cell architecture is shown in Figure 6.1. It has a PowerPC host processor (PPE) and a set of eight computation-specific processors, known as synergistic processing elements (SPEs). The memory subsystem offers private memories for each SPE processing elements and a global memory space, to which only the PPE has direct access. Each SPE has a Memory Flow Controller (MFC) for handling all data transfers. All processors and I/O interfaces are connected by the coherent interconnect bus which is a synchronous communication bus.

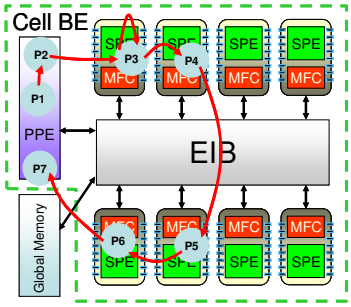

Figure 6.1: A 7-process PPN mapped onto the Cell BE platform.

The mismatch mentioned earlier is illustrated with the example in Figure 6.1. A PPN consisting of 7 processes and 7 FIFO channels is mapped onto the Cell BE platform. Processes $P1$, $P2$ and $P7$ are mapped onto the PPE, and processes $P3$ to $P6$ are mapped onto different SPEs. The FIFO communication channels must be mapped onto the Cell BE communication, synchronization and storage infrastructure. On the one hand, the semantics of the FIFO communication is very simple: Producer and Consumer processes in a producer/consumer pair interact asynchronously with the communication channel to which they are connected. The synchronization between the Producer and the Consumer is by means of blocking read/write on empty/full FIFO channels. On the other hand, in the Cell BE platform the processors are connected to a synchronous communication bus and there is no specific HW support for

blocking FIFO communication. Therefore, the PPN communication model and the Cell BE communication infrastructure do not match. The FIFO channels have to be realized by using the private memory of a SPE, and/or the global memory, and the Cell BE specific synchronization methods which may be costly in terms of communication latency. The challenge is how to do this in the most efficient way, i.e., to minimize the communication latency.

The Intel IXP

The IXP Network processor [1] is built to operate in real-time on internet traffic while being completely programmable. The architecture uses microengines that have hardware multi-threading support and various communication structures to move streams of data around as efficiently and quickly as possible. We focus on the IXP2400 of which a schematic is shown in Figure 6.2.

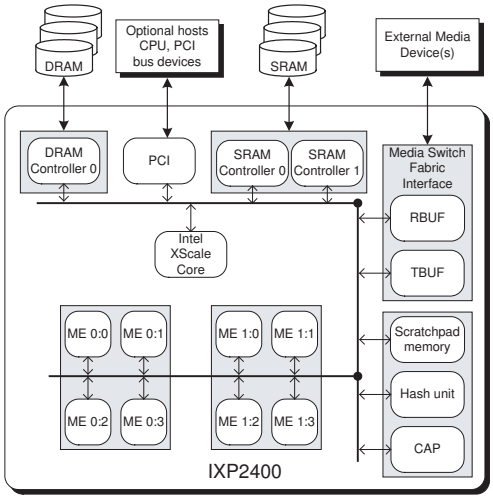

Figure 6.2: IXP2400

The IXP2400 has an Intel XScale Core and eight microengines (ME0.0 - ME1.3) clustered in two blocks of 4. Other relevant parts are the specialized controllers to communicate data with off-chip DRAM and SRAM, the scratch path memory, and the Media and Switch Fabric (MSF) Interface. The MSF interface governs the communication with the Ethernet connection. The IXP2400 can receive and transmit data on this interface at a speed of 2.5Gbps. The XScale core is a RISC general-purpose processor similar to the processing units found in other hardware, including other embedded computers, handhelds and cell phones. The intended use of XScale on the

network processors is for controlling and supporting processes on the microengines where needed.

A microengine is a simple RISC-processor that is optimized for fast-path packet processing tasks. Its instruction set is specifically tuned for processing network data. It consists of over 50 different instructions including arithmetic and logical operations that operate at bit, byte, and long-word levels, and can be combined with shift and rotate operations in a single instruction. Integer multiplication is supported; floating point operations are not. The microengine has special registers to communicate data quickly and efficiently with DRAM and SRAM, its neighbors and local memory. For example, to communicate with neighboring microengines within a cluster, this microengine can uses special hardware support. Via special registers, it can send data to a neighbor and receive data from a neighbor. This could be used to implement FIFO communication, but only for one channel with a very limited buffer capacity. Furthermore, the microengines have access to hardware rings for accessing circular buffers located in the scratchpad and SRAM memories. In Figure 6.2, it can be seen that these buffers are accessed via the bus. Hence, we remark that the IXP has hardware support for FIFO communication (i.e., the available rings), but it is not as dedicated as in the ESPAM platform [60, 61] where each processor has a dedicated communication memory which can be organized as one or more FIFOs.

6.2 Realizing FIFO Communication

In Section 6.1, we have introduced the IXP and Cell processors. Now we show how we map PPNs onto these platforms. This means that each components of the PPN must be expressed in terms of the C language, i.e., a source-to-source translation. These sources can be compiled with the C compiler for the given platform to generate an executable. We focus on the realization of the FIFO communication, because it is the most platform dependent implementation that must use the provided communication infrastructure of the target platform as efficiently as possible. Thus, we indicate the possible mismatch in the PPN model of computation and the target platform. The processes of a PPN are mapped one-to-one onto processing elements. We do not further elaborate on this. Instead, the reader is referred to [50, 58] for more details.

FIFO Communication on The Cell

In mapping PPN processes onto processing elements of the Cell BE platform, different assignments are possible, i.e., processes can be mapped onto the PPE or onto one of the SPEs. This results in different types of FIFO communication channels. For example, in Figure 6.1 processes P_1 (producer) and P_2 (consumer) are mapped onto

the PPE. Therefore, we say that the FIFO channel connecting them is of PPE-to-PPE type. If the producer and the consumer is the same process that is mapped onto a SPE (like process P_3 in Figure 6.1), then we refer to that FIFO channel as a SPE-to-self FIFO channel. Similarly, we identify PPE-to-self, SPE$_i$-to-SPE$_j$, PPE-to-SPE, and SPE-to-PPE types of FIFO communication channels. It is important to define these channel types as all of them require different implementations since different components of the Cell BE platform are involved. To summarize, we identify the following classes of FIFO channels, classified by connection type: *a*) *class self* (PPE-to-self and SPE-to-self), *b*) *class intra* (PPE-to-PPE), and *c*) *class inter* (SPE$_i$-to-SPE$_j$, PPE-to-SPE and SPE-to-PPE).

The first two classes of FIFO channels are easy to implement efficiently, as FIFOs from these classes are realized using just local (for producer and consumer processes) memories and local synchronization is utilized. In the remainder of this section we therefore focus on the *class inter* FIFO channels, which connect the producer and consumer processes mapped onto two different processing elements. The first issue to be addressed is where the memory buffer of a FIFO has to reside. The Cell BE platform provides two memory storages, thus, the buffer can *i)* reside in global memory or *ii)* can be distributed over the local memories of the producer and consumer processes. The advantage of the former approach is the shared memory that is easily accessible (in a mutually exclusive way). The disadvantage, however, is a substantial synchronization overhead. For example, a SPE process with a FIFO channel of type *inter*, should not only compete for the memory resource, but also move the data from the global storage to the local memory prior to computation. The implication of this is an enormous synchronization overhead and we therefore do not consider this as an option to implement the FIFO buffers. For the second approach, i.e., when the memory buffer of a FIFO channel is distributed between local memories, the issue is how to efficiently implement the FIFO semantics. The issue is that the processors need to be synchronized to ensure mutually exclusive access to the FIFO buffer. This processor synchronization is costly and is necessary as the CELL does not provide hardware support for FIFO communication, i.e., the mismatch between the PPN model of computation and the target platform as we mentioned earlier.

In our approach to realize FIFO communication on the Cell and to minimize the number of processor synchronizations, a number of tokens are grouped and send at once, i.e., token packetizing is used. Packetizing decreases the number of DMA data transfers and subsequently it also decreases the number of synchronizations. Determining the packet size is a very important issue, i.e., depending on the process that initiates the data transfers, deadlocks may occur if the packet size is not chosen correctly. We have therefore chosen to use a run-time solution that simply transfers all available generated data. We refer to this solution as the *FIFO pull strategy* which we discuss next. The reader is referred to [58] for a discussion on the FIFO push

strategy and a comparison between the pull and push strategies.

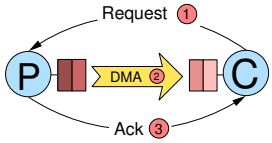

Figure 6.3: Pull strategy for *class inter* FIFO channels.

The FIFO communication between a producer/consumer pair of processes using the pull strategy, consists of 3 steps as shown in Figure 6.3:

1. *Read request* (1). The consumer first tries to read from its local buffer. If it is empty, then it sends a request message to the producer and is blocked on reading awaiting an acknowledgement message from the producer. The request message contains the maximum number of tokens the consumer can accept.

2. *Data transfer* (2). The producer which receives the read request can either be blocked on writing to its local storage or be busy executing a function. If it is blocked, it serves other requests immediately. If it is executing then it immediately serves the request after execution. In either case, the producer handles the request and transfers all tokens it has available for the consumer as one packet by means of a DMA transfer.

3. *Acknowledgement* (3). The producer notifies the consumer after completion of the data transfer issuing a message containing the total number of tokens which have been transferred as one packet in the previous step.

Thus, the *pull* strategy requires two synchronization messages for each DMA data transfer (step (1) and (3)) and the packetizing of tokens is realized in step (2). For every read request of a single data token, the producer sends all its available data to the consumer. Therefore, we refer to this mechanism as dynamic packetizing. The only way to control the dynamic packetizing is by setting the size of the memory buffer, i.e., the larger the size, the larger the packet's size that can be assembled.

FIFO Communication on The Intel IXP

Since the FIFO is such a central element in the IXP, different implementations exist. We have found that six different FIFO types can be realized on the IXP as shown in Figure 6.4. The various realizations make a different trade-off between speed, claimed resources, and size. A short description of the different realizations is given below:

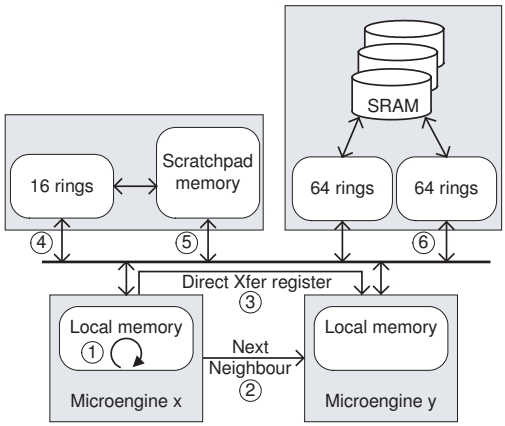

Figure 6.4: FIFO options on the IXP2400

1. *Local memory*. Each microengine has a fast accessible local memory of 640 longwords (2Kb) that is shared among all threads running on that microengine. It can be used to implement a very fast FIFO channel between processes mapped onto the same microengine.

2. *Next-neighbor*. Each microengine has 128 next-neighbor registers. They can be used to implement a very fast, dedicated, FIFO channel between processes mapped onto a limited set of other microengine that are neighbors. The registers can be used in three modes: an extra set of general purpose registers, one FIFO channel of 128 longwords, or as 128 separate registers accessible by the neighboring microengine.

3. *Direct Transfer registers*. Each microengine has 128 SRAM and 128 DRAM registers. They can be used to exchange date with any other process on a microengine. The direct transfer registers use the standard bus and are slower than the previously explained communication mechanism.

4. *Scratchpad rings*. There are 16 sets of special ring registers available on the scratchpad unit. These ring registers provide hardware support to implement the head and tail pointers of a FIFO channel located on the scratchpad memory.

5. *Scratchpad memory*. The ring registers can also be implemented in software directly. These software ring registers implement the head and tail pointers of a FIFO channel located on the scratchpad memory. This is much slower mechanism than the hardware ring register support.

6. *SRAM*. SRAM rings are hardware supported FIFO channel implementations. Each SRAM memory channel has a queue descriptor table which can hold 64 values. Since the IXP2400 has two SRAM memory channels, a total of 128 rings are available.

When very fast FIFO channels are needed, the local memory or next-neighbor registers should be employed. If this is not possible, the hardware support ring registers and scratchpad memory should be used. If this is not possible, the software supported rings should be used. Finally, the SRAM supported FIFO channels should be used. They are the slowest, but can hold the largest amount of data. We implemented a very simple assignment strategy for the processes and FIFO channels of a PPN. FIFO channels are assigned in a greedy way to the fastest possible location. If the FIFO buffer is too large for that location, it is assigned to the next fastest FIFO location.

6.3 Performance Results

We use the techniques presented in Section 6.2 to execute PPNs on the IXP and Cell. We measure the performance results and compare them with results on the dedicated ESPAM platform that is designed to execute PPNs as efficiently as possible.

The Cell

In this section we present several experiments of PPNs mapped onto the Cell platform. The goal is to show the impact of tokens packetizing on synchronization overhead induced in the *class inter* FIFO channels using the *pull* strategy. In addition, we compare the results of two PPNs mapped onto the Cell with the results for the same PPNs mapped onto the ESPAM platform. The Cell experiments are carried out on the `Playstation 3` platform, where the program code has been compiled with IBM's XLC compiler and the `libspe2` library.

In the first experiment we map a JPEG encoder application onto the Cell BE platform. The encoder takes a stream of frames with sizes of 512×512 pixels and applies the JPEG algorithm on these frames. The corresponding PPN consists of 7 processes and 15 FIFO channels. We map the computationally intensive processes DCT, Q and VLE on different SPEs, whereas the other processes are mapped onto the PPE. For this application, buffer sizes of 1 will give a deadlock free network, which means that we can observe token packetizing only when the buffer sizes are increased. Therefore, we run the PPN with four different configurations: we use FIFO buffer sizes of 1, 16, 32 and 48 tokens.

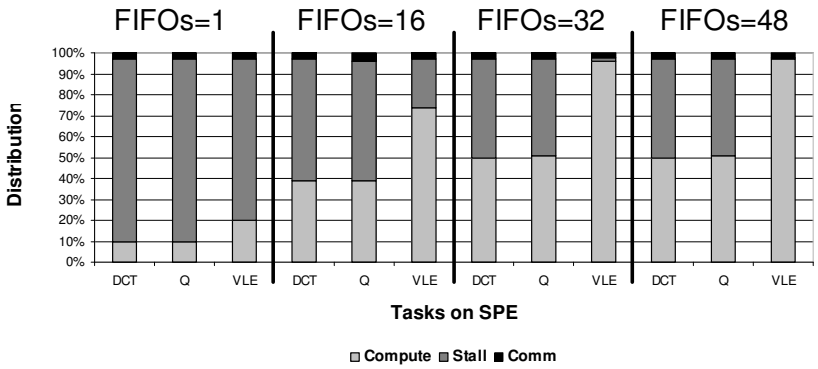

Figure 6.5: Distribution of times the DCT, Q and VLE processes of JPEG encoder spent in computation, stalling and communication for non-packetized and packetized versions.

All bars in Figure 6.5 depict the distribution of the time the DCT, Q and VLE tasks spend in computing, stalling and communicating. Each bar shows how much time processes spend on real computations and thus also how much time is spent in the communication overhead. While stalling, a process is awaiting the synchronization messages from other processes, i.e., showing the synchronization overhead. In the communicating phase, a process is transferring the actual data. The first 3 bars in Figure 6.5 correspond to the configuration with all buffer sizes set to one token. The remaining bars show results of configurations with larger buffer sizes illustrating the effect of token packetizing. We observe a redistribution between computation and stalling fractions in all tasks: the stalling parts have been decreased, while the computation parts were increased. Thus, the packetizing decreases the synchronization overhead. In Figure 6.6, the overall performance of the PPN with different buffer sizes is shown. We observe that the performance increases when the processors spend less time in synchronization.

In four more experiments, we want to investigate the benefits of packetizing in applications with different computation-to-communication ratios. For this purpose, we mapped JPEG2000, MJPEG, Sobel, and Demosaic applications onto the Cell BE. The first two application have coarse-grained computation tasks, while the latter two are communication dominant. For each application, we compare the throughput of the sequential version running on the PPE and two parallel versions: the first one is with minimum buffer sizes that guarantee deadlock free network, i.e., without packetizing possible, and the second, with buffer sizes which are larger than the previous version to allow packetizing. The experiments are depicted in Figure 6.7. Note that the y-axis is a log scale of the throughput in Mbs (mega bits per second).

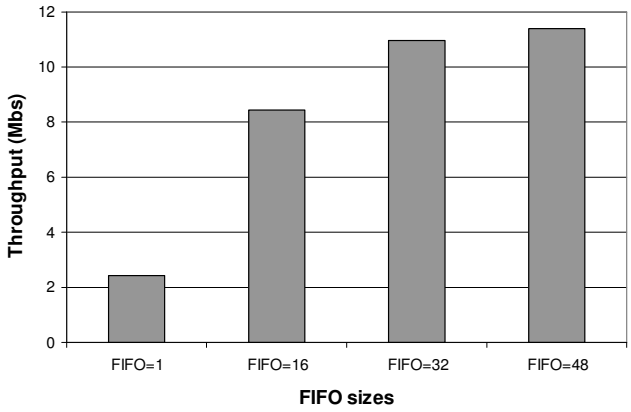

Figure 6.6: Throughput of JPEG encoder with different FIFO sizes

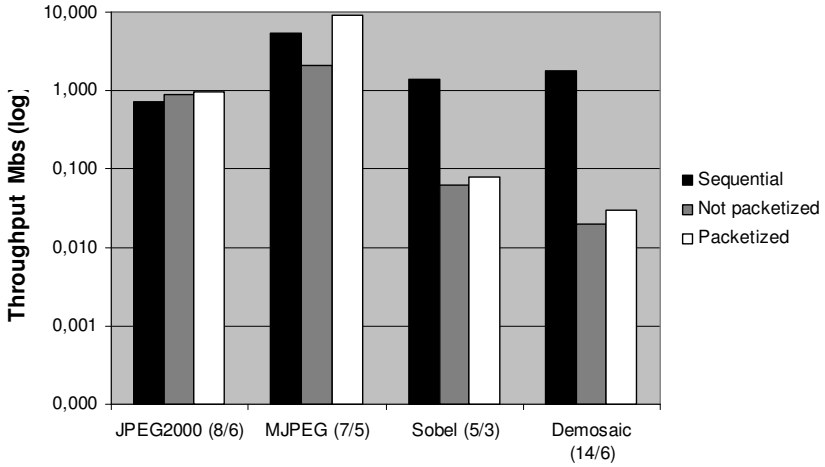

Figure 6.7: Throughput comparison of sequential, non-packetized and packetized versions of JPEG2000, MJPEG, Sobel, and Demosaic applications.

For all algorithms, the packetized versions work better than the non-packetized version. As the JPEG2000 and MJPEG are characterized by their coarse grain tasks, the communication overhead is insignificant and we see that the parallel versions are faster than the sequential version for all, but non-packetized MJPEG algorithms. The Sobel and the Demosaic kernels have very lightweight tasks. Thus, the introduced inter-processor communication and overhead are more costly than the computations themselves. This is the reason the bars in the third and fourth experiments in Figure 6.7 show a significant slow-down compared to the sequential application. The

conclusion is not to consider parallelization for communication dominant applications on the Cell BE platform. We, therefore, ignore the Sobel and Demosaic applications in the comparison of the performance results for applications mapped onto both the Cell and the ESPAM, i.e., we focus on the JPEG2000 and MJPEG applications. The measured performance results for the JPEG2000 application on the Cell and ESPAM are shown in Table 6.1.

Arch.	# clock cycles	CPU freq.	time (sec)
Cell	$288 \cdot 10^6$	3.2 Ghz	0.09
ESPAM	$60 \cdot 10^6$	100 Mhz	0.6

Table 6.1: Measured Performance Results JPEG2000

We observe that the execution time (i.e., the fourth column) is much smaller on the Cell than on the ESPAM platform. This result is mainly due to the clock frequency of the Cell that is a factor of 30 higher than the ESPAM platform. Despite this factor of 30 in the clock frequency, the execution time is *not* 30 times better on the Cell, instead, it is only 7 times better. We observe the same trend for the execution times of the MJPEG application shown in Table 6.2.

Arch.	# clock cycles	CPU freq.	time (sec)
Cell	$1200 \cdot 10^6$	3.2 Ghz	0.375
ESPAM	$300 \cdot 10^6$	100 Mhz	3

Table 6.2: Measured Performance Results MJPEG

The execution time of the MJPEG application is roughly 10 times better on the Cell compared to the ESPAM. Hence, we conclude that the Cell platform is a good platform to obtain low absolute execution time numbers, but it is not necessarily the most efficient platform. The reason is that its clock frequency is 30 times higher than the ESPAM platform (i.e., the Cell is power hungry), but the execution times are not 30 times better. Instead, they are only 10 times better. In other words, the Cell is faster in terms of execution time, but it is not proportional to its much higher clock frequency. The reason is that the FIFO primitives are more costly on the Cell than on the ESPAM platform, i.e., there is more overhead because the Cell communication infrastructure does not support any FIFO communication with hardware components. The reason that the ESPAM platform runs at 100 Mhz, is that it is prototyped on FPGAs. If the ESPAM platform is implemented in ASIC technology as the Cell and the IXP, then the frequency can be higher. As a result, the ESPAM platform would become better in terms of performance than the Cell and the IXP running at a lower frequency, which means that it is also more power efficient.

The Intel IXP

In the experiments for the IXP processes, we consider the QR matrix decomposition algorithm. The corresponding PPN consists of 5 nodes and 12 FIFO channels, see also Figure 5.11. Each process is mapped on a microengine and all FIFO channels are mapped on hardware assisted scratchpad memory rings, i.e., option 4 in Figure 6.4. The reason to map all FIFOs using option 4 is that all FIFOs fit in that memory and that we can actually test the provided hardware support for FIFO communication of the IXP. Note that despite this hardware support for FIFO communication, there is a small software interface implementation that takes care of the FIFO read/write function calls in the read/write phases of the processes. We do not consider the next neighbor link, i.e., option 1 in Figure 6.4, because only 1 FIFO can be mapped and the storage space is limited. Furthermore, we chose not to implement the process functions such that we measure only the FIFO communication in the PPN. Processing a 5x6 version of QR took 40247 cycles on the IXP as shown in the first row of Table 6.3. Note that this measurement only says something about the FIFO communication (read and write phase of a process) as no real function is executed.

Arch.	# clock cycles	CPU freq.	time (μs)
IXP	40247	600 Mhz	67
ESPAM: 5 MB	3865	100 Mhz	39
ESPAM: full HW	213	108 Mhz	2

Table 6.3: Measured Performance Results QR

To assess the efficiency of our FIFO implementation on the Intel IXP processor, we create two ESPAM hardware solutions prototyped on a Xilinx FPGA for the same QR application and compare the performance numbers. We create a platform with 5 Microblaze microprocessors for each process of the PPN, and connect the processors with a dedicated crossbar. The other hardware platform that we create does not use any microprocessors, but all functionality is implemented in hardware, i.e., a full hardware solution. The measured performance results for these two hardware platforms are respectively shown in row 2 and 3 in Table 6.3. It can be seen that the 5 MicroBlaze microprocessor platform executes the QR application in 3865 cycles, while the full hardware implementation executes in 213 cycles. If we take into account the frequencies of the different platforms, i.e., the 3rd column in Table 6.3, then we can compute the execution times that are shown in the 4rd column. These execution times allow a comparison of the QR PPN implementation on 3 different platforms. We observe that the IXP implementation is the slowest, despite the fact that it is running at the highest frequency. We conclude that the more dedicated the communication gets, the higher the performance, i.e., there is roughly a factor of 30 between the ex-

tremes: the software solution on the IXP and the ESPAM full hardware solution. In the IXP we need to share a bus, in the FPGA with MicroBlazes we share a crossbar to communicate between MicroBlazes, and in the full hardware implementation, only dedicated FIFO channels are used to communicate between processes. Moreover, in the IXP there is still some synchronization and control required to handle all FIFO accesses, while in the hardware platforms the producer/consumer pairs are truly decoupled. If we compare the IXP with the 5 MicroBlaze processor ESPAM platform, then the execution time is almost 2 times worse, while the frequency of the IXP is 6 times higher. Thus, Table 6.3 illustrates the penalty that must be paid for mapping PPNs onto a platform that does not support the execution of PPN as efficiently as ESPAM does.

6.4 Discussion and Summary

In this chapter, we showed approaches to execute PPNs on the Cell and IXP platforms, i.e., two commercial-off-the-shelf (COTS), programmable MPSoC platforms. We compared the measured performance results of PPNs executed on these 2 platforms with the performance results obtained on the ESPAM platform. The Cell, IXP, and ESPAM platforms can be characterized as follows: the ESPAM is the most dedicated regarding the execution of PPNs and is prototyped on a FPGA, the IXP is dedicated to streaming data, but is not as dedicated in executing PPNs as ESPAM, and the Cell is the most general purpose compute platform. The platforms run at different frequencies: the ESPAM platform is prototyped on a FPGA and thus runs at 100 Mhz, the IXP runs at 600 Mhz, and the Cell at 3.2 Ghz.

In Section 6.3, we have shown experiments of PPNs executed on these 3 different platforms. Thus, we were able to compare the execution time of the PPNs. From the experiments in Section 6.3, it becomes clear that the IXP processor is not the best platform a designer can select if he/she is free to choose any of these 3 platform as the target platform. Despite the FIFO support in the IXP, the measured execution times of the PPNs are higher than on the dedicated ESPAM platform, while the IXP's clock frequency is 6 times higher than the ESPAM platform. The Cell platform on the other hand, can be a very good platform candidate. Its very high clock frequency compensates the lack of the hardware FIFO support and the overhead caused by the software implemented FIFO communication. However, this overhead makes the Cell not the most efficient platform. While its frequency is 30 times higher than the ESPAM platform, the execution time is only 7 times better. Therefore, we conclude that the Cell is the best platform to obtain the lowest absolute performance numbers. However, the Cell is also the most power hungry solution since it runs at 3.2 Ghz. The frequency of the ESPAM platform can be increased if implemented on the ASIC

technology like the Cell. Then, the ESPAM platform would not only give the best absolute performance results, but it would also be more power efficient. In addition, we remark that PPNs with very light-weight tasks can result in execution times on the Cell that are worse than the sequential version of the application. Again, the reason is the expensive software implemented FIFO communication on the Cell platform. This fact indicates that the designer must carefully take into account the properties of the PPN and the platform in his decision to choose a particular platform. In any case, the ESPAM platform is the most efficient one because it is dedicated to execute PPNs as efficiently as possible.