# Realizing FIFO Communication When Mapping Kahn Process Networks onto the Cell

Dmitry Nadezhkin, Sjoerd Meijer, Todor Stefanov, and Ed Deprettere

Leiden Institute of Advanced Computer Science
Leiden University, Niels Bohrweg 1, 2333CA Leiden, The Netherlands
{dmitryn,smeijer,stefanov,edd}@liacs.nl

**Abstract.** Kahn Process Networks (KPN) are an appealing model of computation to specify streaming applications. When a KPN has to execute on a multi-processor platform, a mapping of the KPN model to the execution platform model should mitigate all possible overhead introduced by the mismatch between primitives realizing the communication semantics of the two models. In this paper, we consider mappings of KPN specification of streaming applications onto the Cell BE multi-processor execution platform. In particular, we investigate how to realize the FIFO communication of a KPN onto the Cell BE in order to reduce the synchronization overhead. We present a solution based on token packetization and show the performance results of five different streaming applications mapped onto the Cell BE.

**Keywords:** Models of Computation, Kahn Process Networks, distributed FIFO communication, the Cell BE platform.

## 1 Introduction

One of the driving forces that motivated the emergence of multi-processor systems on chip (MPSoCs) originates from the complexity of modern applications [1]. Many applications are specified with complex block diagrams that incorporate multiple algorithms. Such applications are called heterogeneous. The emergence of heterogeneous applications led to the design of heterogeneous MPSoC architectures which provide improved performance behavior by executing different algorithms, which are part of an application, on optimized/specific processing components of an MPSoC. However, heterogeneous MPSoCs are very hard to program efficiently, and still it is not very clear how this could be done in a systematic and possibly automated way.

It is a common believe that the key to solve the programming problem is to use parallel models of computation (MoC) to specify applications [2]. This is because the structure and executional semantics of parallel MoCs match the structure and executional semantics of MPSoCs, i.e., a parallel MoC consists of tasks that can execute in parallel and an MPSoC consists of processing components that run in parallel. Nevertheless, in many cases there is a mismatch between the communication semantics of a MoC and the communication infrastructure available in an MPSoC. Therefore, a major issue when programming an MPSoC, is to figure out how to bridge such mismatch, i.e., how to realize the communication semantics of a MoC using the available communication infrastructure of the MPSoC. Unfortunately, a solution approach to the mentioned mismatch is specific for a given MPSoC platform and MoC.

In this paper, we share our experience in bridging the mismatch between the communication semantics of the Kahn Process Network (KPN) model of computation and the communication infrastructure of the Cell BE platform. The Cell BE platform [3] is a very good representative example of a state-of-the-art heterogeneous MPSoC platform. 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 PPE has direct access, while each SPE utilizes accompanied Memory Flow Controller. The processors and I/O interfaces are connected by the coherent interconnect bus which is a synchronous communication bus. The KPN model is a very good representative of the class of dataflow models used to specify streaming applications. A KPN is a graph in which the nodes are active entities (processes or tasks or threads) that communicate point-to-point over FIFO channels. Most of the dataflow models are of the same nature, i.e., they consist of tasks connected with FIFO channels.

The mismatch mentioned earlier is illustrated by the example in Figure 1 where a KPN consisting of 7 processes and 7 FIFO channels is mapped onto the Cell BE platform. Processes $P_1$, $P_2$ and $P_7$ are mapped on the PPE, and processes $P_3$ to $P_6$ are mapped on the SPEs. The FIFO communication channels has to 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 that they are connected to, and the synchronization is by means of blocking read/write. 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 KPN communication model and the Cell BE communication infrastructure do not match. The KPN FIFO channels have to be realized by using

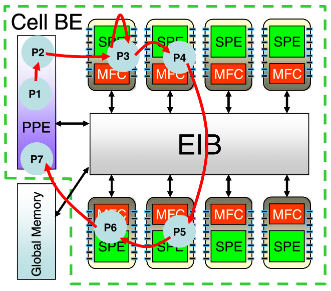


**Fig. 1.** A 6-process dataflow network mapped onto the Cell BE platform

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.

In the following section, we give a survey of related works. In Section 3 we consider the particular issues of realizing FIFO communication semantics of the KPN model of computation on the Cell BE platform. Section 4 illustrates our realization of the FIFO communication channels on the Cell BE platform. In Section 5, we show some experiments with real-world applications. Section 6 concludes the paper.

## 2 Related Work

In a similar work [4], KPNs have been mapped onto the Intel IXP processor. The IXP, however, has hardware support for FIFO buffers and no optimizations have been applied to reduce communication latencies.

Another model-based project that is similar to our approach in programming the Cell BE platform is the architecture-independent stream-oriented language StreamIt [5], which shares some properties with the Synchronous DataFlow (SDF) [6] model of computation. The Multicore Streaming Layer (MSL) [7] framework realizes the StreamIt language on the Cell BE platform focusing on automatic management and optimization of communication between cores. All data transfers in the MSL are explicitly controlled by static scheduler, and thus, a synchronization in FIFO communication is not an issue. However, this approach is limited to applications that can be specified with SDF. Our approach is more general as a broader class of applications can be specified with KPNs compared to SDF, at the cost of introducing blocking read and write FIFO primitives, ensuring that processes block if data is not available or cannot be written. The introduced synchronization becomes an issue, which we tackle in this paper.

As to the low-level communication on the Cell, MPI like The Cell Messaging Layer[8] library is implemented guided by the similar idea as in our approach, i.e., receiver-initiated communication. However, the library offers just low-level `_send` and `_receive` primitives without focusing on realization of FIFO abstraction.

## 3 Issues of Mapping KPNs onto the Cell BE Platform

In mapping KPN processes onto processing elements of the Cell BE platform, different assignment options are possible: each processor can host one or more KPN tasks. For the PPE processor which has two hardware threads and runs a multitasking operating system, a threaded library can be utilized to host several KPN tasks. Although multitasking for the SPEs is also possible, in practice it is inefficient as the context switching is very expensive: all the code and data, while switching, should be saved in the global memory. Thus, in this paper we consider that only one KPN process can be assigned to each SPE processor.

Given the considerations above, there is a variety of mapping strategies which lead to the appearance of different types of FIFO communication channels. For example, in Figure 1 processes $P_1$ (producer) and $P_2$ (consumer) are mapped onto the PPE, and we say that the FIFO channel connecting them is of PPE-to-PPE type. If the producer and the consumer are one and the same process mapped onto the SPE (like process $P_3$ in Figure 1), then we refer to the FIFO channel as of SPE-to-self type. Similarly, we identify PPE-to-self, SPE$_i$-to-SPE$_j$, PPE-to-SPE, and SPE-to-PPE types of FIFO communication channels. All of them require different implementations as different components of the Cell BE platform are involved. Thus, 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 memories and synchronization primitives. We will not discuss the detailed implementation of these FIFOs in this paper, however, we briefly explain the realization. In the *class self*, the FIFO channel connects a process with itself. Since there is only one thread of control, the access to the FIFO is ordered and therefore no special synchronization is required. In the *class intra*, where producer and consumer processes are mapped on the PPE, a FIFO channel is a shared resource in shared memory to which mutual exclusive access is applied. We rely on the pthread library to deal with this producer/consumer communication.

In this paper we focus on the *class inter* FIFO channels, which connect the producer and consumer processes mapped onto two different processing elements of the Cell BE platform. 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 reside in global memory or be distributed partly between private memories of the producer and consumer processes. The cons of the former approach lie in the presence of the shared component, which should be accessed with mutually exclusive pattern. For example, a SPE process connected to *class inter* FIFO, 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 enormous synchronization overhead making the performance of this approach not better than the performance of the sequential version of an application.

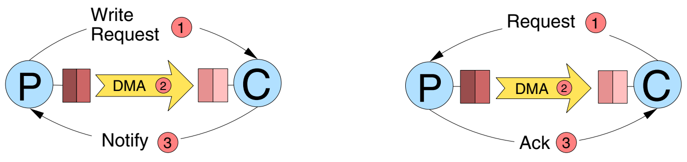
When the memory buffer of a FIFO channel is distributed between private memory storages of a producer and consumer processes, the issue is how to implement the FIFO semantics over the distributed memory buffer such that it does not mask the performance benefits of going distributed. The FIFO semantics is realized by means of Direct Memory Access (DMA) transfers and synchronization messages between producer and consumer processes. The more a KPN is communication dominant, the more synchronization overhead is generated which can lead to a performance penalty. Therefore, the issue is to minimize the number of data transfers over the distributed FIFO channels as much as possible.

## 4 Solution Approach

Our approach in minimizing the DMA data transfers is based on packetizing of tokens. In this case, a number of tokens are grouped into a single packet, which is transferred as one DMA transfer. Packetizing decreases the number of DMA data transfers or in other words, it decreases the number of synchronizations. Determining the packet size becomes a very important issue, and as it will be shown, it depends on how the DMA data transfers are initiated. Also, it will be shown, that in some cases an incorrect packet size may lead to a deadlock.

Before determining the size of a packet, we need to consider the possible protocols for realizing the FIFO semantics over the distributed memory buffer in detail. As all FIFO channels we consider are point-to-point, tokens can be transferred either in a data driven or a data demand fashion. The former case follows a *push* strategy in which the producer initiates a data transfer as soon as it has produced data, whereas the latter case follows a *pull* strategy in which the consumer initiates a data transfer as soon as it requires data. The two strategies are shown in Figure 2, where the numbered circles indicate the order of synchronization messages along with the DMA data transfer; nodes P and C represent the producer and the consumer, respectively. We explain both strategies in detail and discuss pros and cons, and provide our solution choice.

In the *push* strategy depicted in Figure 2a, the producer first makes a write request as soon as one or more tokens have been produced (1). Then, the consumer transfers the data with a DMA (2) and sends the notification message to the producer (3). Thus, two synchronization messages are required to complete one DMA data transfer. Packetizing of tokens happens on the producer side and the size of the packet should be known before the data transfer. However, wrong computed size may result in a deadlock in some network topologies. For example, consider Figure 3. The network consists of 3 tasks ($P_1$, $P_2$ and $P_3$) and 3 FIFO channels ($F_1$, $F_2$ and $F_3$). Assume that for channel $F_1$, the packet size, which guarantees deadlock free network evaluation equals to 3. Then, we change the packet size of $F_1$ to 4 tokens. When $P_1$ has generated 3 tokens, instead of sending them to $P_2$, $P_1$ continues to produce new tokens to fill a packet, reading from the input channel $F_3$. The $P_2$ process cannot proceed, as packet from $P_1$ has not been sent because the packet is not complete, and the data is not available. Similarly, the $P_3$ process gets blocked in reading data from $P_2$, and thus, it cannot produce the token for $P_1$. The network is in a deadlock. Hence, for the



(a) push (b) pull

**Fig. 2.** Push and pull strategies for *class inter* FIFO channels

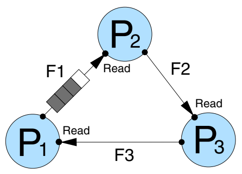


**Fig. 3.** An example of a deadlock in the *push* strategy

*push* strategy the safe size of a packet for all FIFO channels should be computed at compile time.

The *pull* strategy for realizing FIFO semantics over the distributed memory buffer is composed of the following three steps shown in Figure 2b:

1. *Read request* (1). The consumer first tries to read from its local buffer. If this buffer does not contain the required data, then it sends a request message to the producer and gets blocked on reading the acknowledgement message from the producer. The request message contains the maximum number of tokens the consumer may 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, and if it is executing then it immediately serves the request after execution. So in any 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.

In the *pull* strategy, for every DMA data transfer, also two synchronization messages are required and the size of the packet to be transferred is computed dynamically in step (2) of the protocol given above. The only way we can control the dynamic packetizing is by setting the size of the memory buffer. The larger the size, the larger packet can be assembled. Since a consumer gets the data as soon as it is available, a deadlock is impossible in the *pull* strategy.

Both strategies have their own advantages and disadvantages. On the one hand, in the *push* strategy, the required computation of packet sizes at compile time is not always possible for any KPN. This is because, the rate of production and consumption of tokens in channels might not be known at compile time. In the *pull* strategy, the computation of packet sizes is dynamic, i.e., computed at run-time, hence always possible. On the other hand, the *push* strategy is predictable, i.e., packet sizes are know at compile time and this information can be used to reason about performance, while in the the *pull* strategy such reasoning is not possible. Moreover, in some networks, the dynamically computed packet sizes may not be larger than one token. A simple example of such network is a producer/consumer pair where the rate of token production and consumption is the same. Regarding the synchronization overhead, both strategies require the same number of synchronization messages for a single DMA data transfer. Based on the above comparison between the *push* and the *pull* strategies, we have chosen the *pull* strategy as it is generic and can be used for any KPN.

## 5 Experimental Evaluation

In this section we present several experiments of KPNs mapped onto the Cell platform. The main goal is to show the impact of tokens packetizing on synchronization overhead induced in the *class inter* FIFO channels using the *pull* strategy.

To carry out the experiments, we have developed a tool named Leiden Cell C-code Generator (LCCG), which performs a mapping of a KPN specification onto the Cell BE platform in an automated way. The tool accepts KPNs generated by the `pn` and the Compaan compilers [9,10]. Given a KPN specification and a mapping file in which processes are assigned to processors, LCCG generates `C`-code for the PPE and SPE processor elements, as well as specific FIFO read and write primitives for each type of communication channel. After running the LCCG tool, the generated code is compiled on the `Playstation3` platform with IBM's XLC compiler using the `libspe2` library.

### Experiment: JPEG Encoder

In this 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 KPN is depicted as a graph in Figure 4. The KPN consists of 7 processes and 15 FIFO channels. Each task of the application corresponds to a node in the graph; every channel is annotated by a name of a data structure which specifies a token, and FIFO sizes which guarantee deadlock free execution of the network. 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

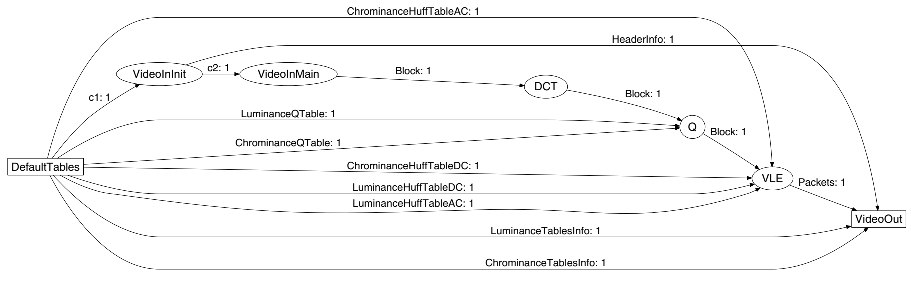


**Fig. 4.** KPN specification of the JPEG encoder

by increasing the buffer sizes. Therefore, we run the KPN with four different configurations: we use FIFO buffer sizes of 1, 16, 32 and 48 tokens.

All columns in Figure 5a depict the distribution of the time the `DCT`, `Q` and `VLE` tasks spend in computing, stalling and communicating. It shows how much time processes spend on real computations and thus also how much time is spend 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 5a 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. The overall performance of different versions is depicted in Figure 5b. We observe that as the processors spend less time in synchronization, the performance increases.

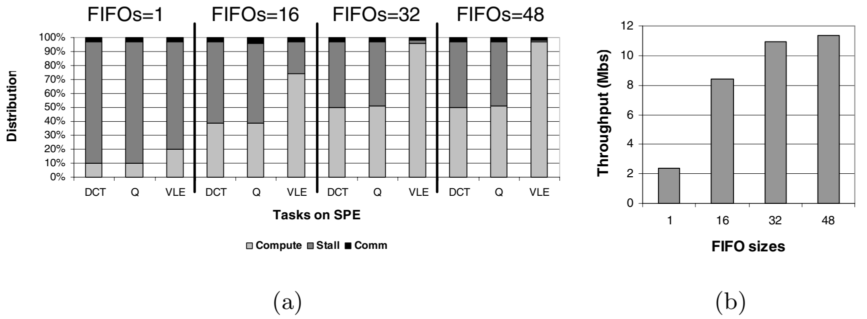


**Fig. 5.** Results of experiments with JPEG encoder: a) distribution of times the `DCT`, `Q` and `VLE` processes of JPEG encoder spend in computation, stalling and communication for non-packetized and packetized versions; b) throughput of JPEG encoder with different FIFO sizes

## Other Experiments

In other experiments we investigated the benefits of packetizing in applications with different computation-to-communication ratio. For that 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 compared 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 then the previous version to allow packetizing. The experiments are depicted in Figure 6. The y-axis is a log scale of throughput in Mbit/s.

For all algorithms, the packetized versions work better than non-packetized. As the JPEG2000 and MJPEG are characterized by their coarse grain tasks, the

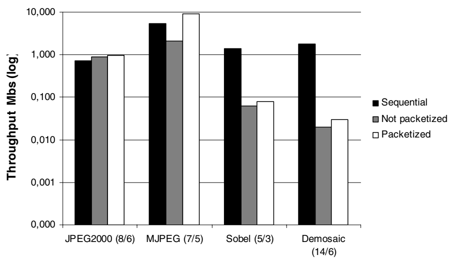


**Fig. 6.** Throughput comparison of sequential, non-packetized and packetized versions of JPEG2000, MJPEG, Sobel, and Demosaic applications

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 itself. This is the reason the columns in the third and fourth experiments in Figure 6 show a significant slow-down compared to the sequential application. The conclusion is not to consider fine-grained parallelization of applications on the Cell BE platform using FIFO communication.

## 6 Conclusion

In this paper, we presented a solution for bridging the mismatch between the KPN communication model and communication primitives of the Cell BE platform. The absence of hardware support for FIFO communication in the Cell BE, makes reading and writing from/to FIFO channels expensive operations. We have investigated several approaches to realize the FIFO communication on the Cell. As a result of our investigation we selected an approach called the *pull* strategy which is based on packetizing of tokens. The experimental results show that this approach gives a performance that is always better than the performance of a network without packetization. All types of FIFO communication channels have been implemented in a tool which automatically generates `C`-code for the KPN tasks and FIFO channels, or in other words, automatically maps a KPN specification onto the Cell BE platform.

**Acknowledgments.** We would like to thank Bart Kienhuis and Hristo Nikolov for the useful discussions on this paper and its results.

## References

1. Wolf, W., Jerraya, A.A., Martin, G.: Miltiprocessor System-on-chip (mpsoc) Technology. IEEE Transactions on Computer-Aided Desing of Integrated Circuits and Systems 27(10) (2008)
2. Martin, G.: Overview of the mpsoc design challenge. In: DAC 2006: Proceedings of the 43rd annual conference on Design automation, pp. 274–279. ACM, New York (2006)

3. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM J. Res. Dev. 49(4/5), 589–604 (2005)
4. Meijer, S., Kienhuis, B., Walters, J., Snuijf, D.: Automatic partitioning and mapping of stream-based applications onto the intel ixp network processor. In: SCOPES 2007: Proceedings of the 10th international workshop on Software & compilers for embedded systems, pp. 23–30. ACM, New York (2007)
5. Thies, W., Karczmarek, M., Amarasinghe, S.P.: Streamit: A language for streaming applications. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)
6. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Computers 36(1), 24–35 (1987)
7. Zhang, X.D., Li, Q.J., Rabbah, R., Amarasinghe, S.: A lightweight streaming layer for multicore execution
8. Pakin, S.: Receiver-initiated message passing over rdma networks. In: IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008, pp. 1–12 (April 2008)
9. Verdoolaege, S., Nikolov, H., Stefanov, T.: pn: a tool for improved derivation of process networks. EURASIP Journal on Embedded Systems, Special Issue on Embedded Digital Signal Processing Systems 2007 (2007)
10. Kienhuis, B., Rijpkema, E., Deprettere, E.F.: Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In: Proc. 8th International Workshop on Hardware/Software Codesign (CODES 2000), San Diego, CA, USA, May 3-5 (2000)