Estimation and Optimization of the Performance of Polyhedral Process Networks

Haastregt, S. van

CHAPTER 3

SYNTHESIZING PPNS

In Chapter 2, we introduced the Polyhedral Process Network model of computation and the PNGEN tool flow which automatically derives PPNs from sequential static affine nested loop programs written in C. We then introduced the ESPAM tool which employs the LAURA model to obtain synthesizable RTL implementations of PPNs. In this chapter, we focus on optimizing the RTL in the aforementioned tool flow. We first investigate shortcomings of the current state-of-the-art techniques and then propose extensions to facilitate more efficient RTL implementations.

3.1 Motivation & Contributions

When implementing industrially relevant applications, such as the sphere decoder application discussed in Chapter 6, and when applying transformations discussed in Chapter 5, we encountered four limitations of the LAURA model and the ESPAM tool. These limitations comprise characterization of functions, incorporation of novel front-end optimizations, handling of more complex domains, and handling out-of-order communication. In this chapter, we present solutions to these four limitations.

First, in the original work describing the LAURA model, only the delay metric of an IP core was considered [ZSKD03, NSD08a]. Such a simplified characterization does not suffice when integrating IP cores generated by HLS tools or when reasoning about system composition. In Section 3.2, we therefore present a more elaborate characterization of IP cores.

Second, the PN tool performs several optimizations that were not taken into account in the original LAURA model. In Section 3.3 and 3.4, we show how data reuse and sticky FIFO optimizations can be leveraged in the LAURA model to obtain more

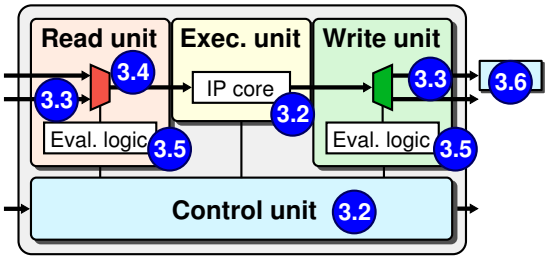


Figure 3.1: Position of the contributions of this chapter in the LAURA model.

efficient implementations.

Third, for complex iteration domains, the evaluation logic of a LAURA processor may become part of the critical path limiting the maximum achievable clock frequency of a system. As a result, the overall throughput of the system is limited. In Section 3.5, we investigate two different approaches to reduce the degradation of the maximum achievable clock frequency.

Fourth, applications with reordering communication could not be implemented using the ESPAM tool. Moreover, the known reordering buffer implementations suffered from read and write penalties with regards to non-reordering buffers [ZTKD02]. In Section 3.6, we present a new reordering buffer design with single-cycle read and write latencies that has been integrated in ESPAM. The particular design enables effortless integration in ESPAM-generated MPSoCs with point-to-point communication. In Section 3.7, we summarize this chapter. The positions of the contributions to the LAURA model have been indicated in Figure 3.1.

3.2 IP Core Characterization

The original LAURA model assumes that the IP core that is integrated into the execute unit comes from an external library. Such a library contains IP cores for different functions and possibly multiple IP cores for the same function that differ in performance and resource cost metrics. Being able to characterize an IP core in a concise way is important when considering performance estimations of PPNs in Chapter 4. To systematically distinguish between different IP cores which possibly implement the same function, we introduce the notion of a function implementation.

Definition 3.1 (Function Implementation)**.**
A *function implementation* is a particular implementation of a process function F. A function implementation is characterized by

- a latency Λ_F and
- an initiation interval II_F,

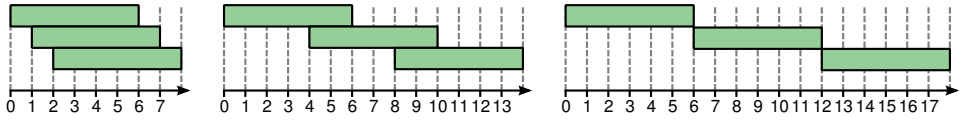
where $\Lambda_F \in \mathbb{N}^+$ is the input-to-output delay in clock cycles, and $II_F \in \mathbb{N}^+$ is the initiation interval in clock cycles.

The delay Λ_F represents the time between the start of a function execution and the moment at which all output has been produced. In Figure 3.2c, we show a time line of three sequential executions of a function implementation with $\Lambda_F = 6$.

The *initiation interval* II represents the amount of time between successive starts of a function implementation. Figure 3.2a depicts a function implementation with $II_F = 1$, allowing an execution of a function to be started every clock cycle. As a result, different executions of the function overlap in a pipeline fashion. Figure 3.2b depicts a function implementation with $II_F = 4$, allowing an execution to be started only every four clock cycles. The amount of overlap between different executions is less than the previous scenario. Figure 3.2c depicts a function implementation with $II_F = \Lambda_F = 6$, resulting in fully sequential executions of the function. This scenario resembles a non-pipelined function implementation. In this thesis, we set $II_F = \Lambda_F$ to model an implementation on a programmable processor on which no overlapped execution of function invocations occurs. A low II implies that the function implementation can deliver a high throughput. However, a low II reduces the opportunities for resource sharing inside a function implementation, resulting in higher resource cost compared to function implementations with a higher II. As such, the II is a key tool in trading off throughput and resource cost of the function implementation.

3.2.1 IP Core Integration

The function implementations in the IP core library may originate from various sources. The corresponding IP cores may be implemented in RTL manually, or the RTL can be automatically derived from a high-level specification using HLS tools. We have successfully implemented IP cores generated by the PICO [Syn10], AutoESL [Xil11], and DWARV HLS tools [YBK+07]. The RTL generated by PICO and



a) $II_F = 1$. b) $II_F = 4$. c) $II_F = \Lambda_F$.

Figure 3.2: Different initiation intervals for an IP core with delay $\Lambda_F = 6$.

AutoESL can be integrated in a straightforward way by connecting the clock, reset, enable, and data ports to the execute unit [HK09]. The RTL generated by DWARV assumes a shared memory model which is different from the distributed memory model employed in the PPN context. Therefore, integrating DWARV cores requires an additional wrapper which transfers data to and from a memory that connects to the DWARV core [NHS+11].

HLS tools such as PICO or AutoESL characterize a generated fixed-latency core by its latency Λ and initiation interval II [Fin10]. In the original LAURA model, only the latency was taken into account and the II value was assumed to be one. To integrate a fixed-delay IP core characterized by Λ and II values, we have extended the LAURA model to take IP cores with $II > 1$ into account. Both Λ and II are incorporated in the control unit of the generated LAURA HDL. Using the delay value, the control unit enables the write unit at the appropriate times, that is, when valid data is produced by the execute unit. Using the II value, the control unit enables the read unit only at valid II boundaries.

Function implementations with a variable delay cannot be characterized accurately by a single number. Instead, a designer may choose to set Λ to the average or worst-case delay value for performance analysis purposes. When integrating a variable-delay IP core, the values Λ and II are not taken into account in the LAURA HDL. Instead, the control unit requires the IP core to indicate when it is ready to accept or produce data.

3.3 Data Reuse

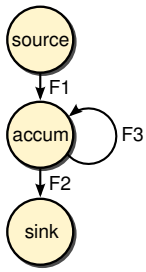
In applications such as filters, often a variable or array element is written once and subsequently read multiple times. For example, the array element `a[1]` in Figure 3.3a is written once when $i = 1$ and read when $j = 1$ (for argument `a[j]`) and $j = 2$ (for argument `a[j-1]`). In a PPN derived from the C code, both reads of `a[1]` are performed by the *accum* process. For the relation from *source* to *accum*, the compiler detects *data reuse*, which means the same token is read more than once from this relation.

A PPN derived from the C code using PNGEN is shown in Figure 3.3b. Channels *F1* and *F3* implement the data reuse channel pair for the relation from *source* to *accum*. Channel *F1* is a regular FIFO which transfers a token when *accum* needs it for the first time. Channel *F3* is a regular FIFO which propagates the token to subsequent iterations of *accum*.

In Figure 3.4, we depict part of a LAURA processor for the *accum* process of Figure 3.3c. Its read unit contains two multiplexers. The lower multiplexer passes tokens

```
for (i=0; i<5; i++) {
  source(&a[i]);
}
for (j=1; j<5; j++) {
  accum(a[j], a[j-1], &b[j]);
}
for (k=1; k<5; k++) {
  sink(b[k]);
}
```

a) C code.



b) PPN with data reuse.

Figure 3.3: A program with data reuse.

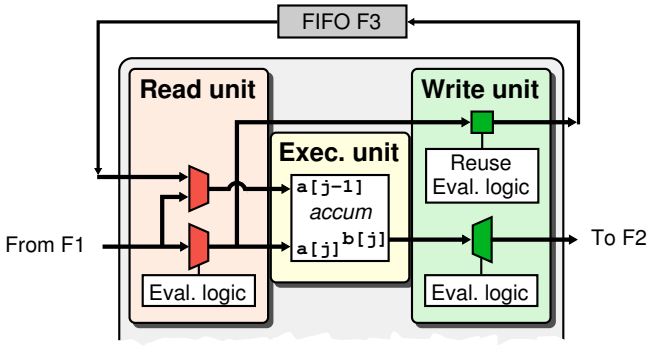


Figure 3.4: Handling data reuse in a LAURA processor.

from FIFO *F1* to the first input of the *accum* IP core. The upper multiplexer selects between FIFO *F1* that is read during the first iteration and FIFO *F3* that is read during subsequent iterations, and passes the token to the second argument of the IP core. The write unit contains a single demultiplexer which propagates the IP core output to FIFO *F2*. To handle the reuse, we extend the write unit with another output port connected to FIFO *F3*. The output port is driven by the first input to the IP core. A separate reuse evaluation logic block ensures that only tokens that need to be propagated to subsequent iterations are written to *F3*. The reuse evaluation logic block duplicates the expressions from the write unit's evaluation logic for the reuse ports to select the correct output port. Tokens that are reused in subsequent iterations can be written to *F3* immediately after reading them, irrespective of the IP core latency. We therefore connect the counters of the read unit to the reuse evaluation logic block.

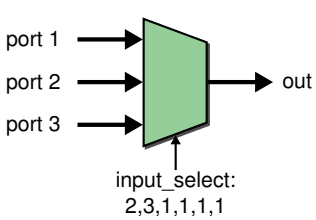
3.4 Sticky FIFOs

As an optimization of data reuse, PNGEN can classify a data reuse channel pair as a sticky FIFO. If the same token is transferred over a FIFO to multiple subsequent iterations of a process, then PN classifies the FIFO as a sticky FIFO and removes the selfloop. During a regular read operation on a sticky FIFO, the receiving process stores the token in a register. Subsequent iterations that need the same token then read from the register instead of the FIFO. This reduces inter-process communication and the number of write operations the producing process has to perform.

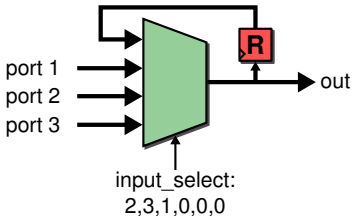
We implement a sticky FIFO by replacing the read multiplexer of a function argument with a "sticky read multiplexer". In Figure 3.5, we illustrate both types of read multiplexers. Figure 3.5a depicts the situation where all of the three input ports of the read multiplexer are connected to regular FIFOs. The read unit's evaluation logic block drives the input_select port of the multiplexer. The output of the multiplexer is propagated to the execute unit. In the example of Figure 3.5a, we first read a token from port 2, then a token from port 3, and then four tokens from port 1, as indicated by the sequence below the input_select port.

Figure 3.5b depicts the situation where port 1 is connected to a sticky FIFO. The output of the multiplexer is both propagated to the execute unit and written into register **R**. The output of register **R** is an additional input to the multiplexer. This additional input is selected when input_select is set to zero. This is illustrated by the sequence below the input_select port. We first read a token from port 2, then a token from port 3, and then a token from port 1. Then, input_select is set to zero which means we reuse the token read from port 1 that is still in **R**. As a result, the process writing to port 1 has to write the token only once.

Since the register is connected to the output of the multiplexer, it also stores tokens read from other ports that can be connected to any type of channel. However, tokens from non-sticky FIFOs are never read from the register, since the semantics of a



a) Port 1 connects to a regular FIFO.



b) Port 1 connects to a sticky FIFO.

Figure 3.5: Read multiplexer architecture.

sticky FIFO ensure that a regular read access is always performed before the token in the register is reused. For the example of Figure 3.5b this means that a zero in the input_select sequence is always preceded by a one, potentially with more zeros in between. Therefore, we do not need a separate register for each sticky FIFO port, but use a single register connected to the multiplexer output.

3.5 Evaluation Logic Optimizations

The main purpose of a LAURA processor is to route tokens from different process ports to the IP core during the appropriate process iterations. The evaluation logic blocks of a LAURA processor select the process ports that are accessed during a given iteration. The evaluation logic is driven by a set of cascaded counters that iterate through the points of the process iteration domain. At each iteration point, an expression is evaluated for each process port. When the expression evaluates to true, the port is accessed in the current iteration. The result of the evaluation is forwarded to the read multiplexer or write demultiplexer of the LAURA processor. In Figure 3.6, we illustrate the internal structure of the evaluation logic by considering the read unit's evaluation logic of Figure 2.12 in more detail. Only one counter is present, because the domain of the process is one-dimensional. The evaluation logic contains an expression for each of the two input ports. Port 1 is accessed during the first five iteration points, as denoted by the bit string in the right part of Figure 3.6. Port 2 is accessed during the remaining five iteration points.

We have identified two problems with the evaluation logic of a LAURA processor. First, the evaluation logic may affect the maximum achievable clock frequency of a LAURA processor, as the expressions become part of the critical path. Second, expressions containing for example `max` or `div` operators are nontrivial to implement. These problems becomes apparent when considering the scheduling transformation discussed in Section 5.1.4, as illustrated in for example Figure 5.11.

We address the first problem by pipelining the evaluation logic, as discussed in Section 3.5.1. We address the second problem by implementing the evaluation logic using ROM tables, as discussed in Section 3.5.2.

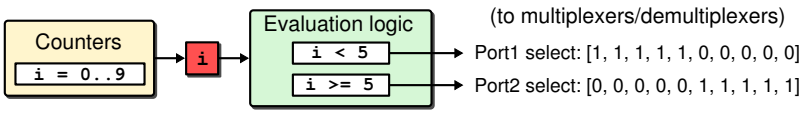


Figure 3.6: Evaluation logic block of a LAURA processor.

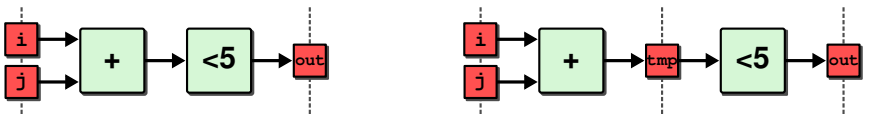
3.5.1 Pipelined Evaluation Logic

To achieve a higher clock frequency, we break long combinational paths into shorter combinational paths that are connected by registers. In Figure 3.7, we illustrate this for the expression $i + j < 5$. Without pipelining, the maximum combinational path length is two because the comparison is connected directly to the addition. In Figure 3.7b, we insert a register between the addition and the comparison. As a result, the maximum combinational path length is reduced to one and therefore the clock cycle period for this circuit can be decreased. However, the evaluation of the expression now takes two clock cycles. Only if subsequent evaluations can execute in an overlapped fashion then a throughput rate of one operation per clock cycle can be sustained at a clock frequency that is higher than the original clock frequency.

The advantage of this solution is that the maximum clock frequency of a LAURA node can be increased at the expense of only a small amount of registers. A disadvantage of this solution is that deciding the amount and insertion points of registers is a non-trivial task. Moreover, control dependencies inside the LAURA model and control dependencies between LAURA processors and other processing or communication components of a system do not allow for unlimited insertion of registers. We have found that pipelining the evaluation logic by one level is still possible.

3.5.2 ROM-Based Evaluation Logic

To implement any non-parametric evaluation logic, we can always resort to a table based implementation. We obtain this table by evaluating all expressions at compile-time and storing the results in a Read-Only Memory (ROM). This technique has already been presented by Derrien et al. [DTZ+05], but was not available in the Daedalus design flow. Derrien et al. already found that ROM based evaluation logic is more expensive in terms of resources than expression based evaluation logic. When realizing designs, we favor expression based evaluation logic, and only use ROM based evaluation logic when expression based evaluation logic requires operators like `max` and `div`, as these operators are not trivial to implement in RTL. Within Daedalus, we can select per processor whether to use expression based evaluation logic or ROM



a) Original: path length of 2 operations. b) Pipelined: max. path length of 1 operation.

Figure 3.7: Expression pipelining.

based evaluation logic.

For each iteration in the process domain, the ROM contains a word that specifies which ports need to be accessed. In a straightforward implementation of ROM-based evaluation logic, all port selection signals for each iteration of the process domain are stored in a table E. For a read or write unit of a process p connected to n ports, such a table E requires

$$n \cdot |D_p| \tag{3.1}$$

bits, where $|D_p|$ is the cardinality of p's process domain. However, many streaming applications exhibit repeating patterns in the ports accessed during subsequent iterations. Like [DTZ+05], we compress such repetition by applying a run-length encoding on the ROM data. This requires an additional table R containing the repetition count of each word in table E.

In Figure 3.8, we show the read unit's evaluation logic of Figure 2.12 implemented using ROM containing run-length encoded port selection patterns. Contrary to Figure 3.6, the evaluation logic block now contains two ROMs instead of a set of expressions. The first ROM shown at the bottom of the evaluation logic block contains table E. A column in this ROM represents the ports that are selected during a set of subsequent iterations. For example, the first column contains the sequence $[1, 0]^T$, meaning the first port is selected while the second port is deselected. The second ROM shown at the top of the evaluation logic block contains table R. It specifies the amount of times each column in E has to be repeated. In Figure 3.8, table R contains $[4, 4]$, meaning that both columns in E should be repeated four times. Thus, the first column is considered in total five times, and then the second column is considered five times. At run time, this results in port 1 being accessed five times, followed by port 2 being accessed five times, as illustrated by the bit strings at the right part of Figure 3.8.

The resource cost of a compressed ROM-based evaluation logic block mainly depends on the sizes of tables E and R. The size of E depends on the number of entries and the number of ports. The size of R depends on the number of entries and the number of bits required to store the largest repetition count occurring in R. This

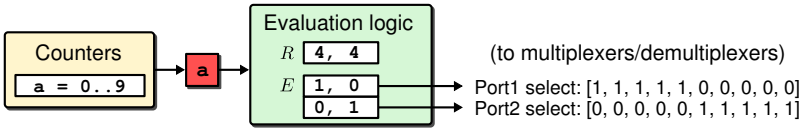


Figure 3.8: Evaluation logic block of a LAURA processor implemented using ROM.

Process - Unit	n	$\lvert D_p \rvert$	$\lvert R \rvert$	$\max(R)$	ROM Size (bits)		
					Uncompr.	Compressed	%
zero-Wr	2	28	13	5	56	65	+16
read-Wr	2	147	42	5	294	210	-29
vectorize-Rd	4	147	42	5	588	294	-50
vectorize-Wr	3	147	42	5	441	252	-43
rotate-Rd	5	441	231	4	2205	1848	-16
rotate-Wr	4	441	212	4	1764	1484	-16
sink-Rd	2	28	13	5	56	65	+16

Table 3.1: Individual ROM sizes for QR decomposition with $K = 21, N = 7$.

yields a total ROM size of

$$|R| \cdot n + |R| \cdot w \tag{3.2}$$

bits, where n is again the number of ports and $w = \lceil \log_2 \max(R) \rceil$. The size of a compressed evaluation logic block may be larger than the size of an uncompressed evaluation logic block in case

$$n \cdot |D_p| \; < \; |R| \cdot n + |R| \cdot w. \tag{3.3}$$

To assess whether this occurs in practice, we consider the QR decomposition application which exhibits complex port selection patterns that reduce compression effectiveness.

In Table 3.1, we show statistics for the individual ROMs of the five processes constituting a QR decomposition application. For example, the third row corresponds to the read unit for the vectorize process. An uncompressed ROM for the vectorize read unit requires $3 \cdot 147$ bits according to Equation (3.1). The compressed ROM requires $42 \cdot 4 + 42 \cdot \lceil \log_2(5) \rceil$ bits according to Equation (3.2). Applying the compression technique to the "zero" and "sink" processes results in ROM sizes that are larger than the sizes of their uncompressed counterparts. This can be attributed to the small domain sizes of these processes. Because each pattern is repeated at most twice, the overhead of table R outweighs the benefits of a smaller number of entries in E.

In Table 3.2, we show the total ROM size with and without using compression for instances of the QR decomposition application. In all cases except the first, the compression technique leads to reduction of the memory cost. For larger values of parameters K and N, the iteration domain sizes of the processes increase. This results in a larger reduction, because the number of additional bits required to store

Parameters		Uncompressed	Compressed	Reduction
K	N	(bits)	(bits)	(%)
3	3	222	226	+1.8
4	4	400	386	-3.5
21	7	5404	4218	-21.9
16	8	5328	3748	-29.7
16	16	20128	8820	-56.2
64	16	78880	34116	-56.7
256	64	4800640	685316	-85.7

Table 3.2: Total ROM sizes for different QR decomposition instances.

higher repetition counts increases more slowly than the number of additional points in the iteration domain.

The worst case for which run-length encoding does not yield any gains is when alternating between two ports. In such a case, the ROM size approaches $n \cdot |D_p|$ bits. The cost of repetition count table R should be added to this, yielding a "compressed" ROM whose size may exceed the size of the uncompressed ROM. However, alternating port selection patterns can often be handled easily using LAURA's conventional expression-based evaluation logic. Therefore, we do not need a ROM-based solution for such cases.

3.5.3 Related Work

All case studies conducted in this dissertation (cf. Chapter 5), the evaluation logic could be successfully implemented in either a pipelined or a ROM-based fashion. However, for applications demanding a clock frequency close to the platform limits, neither a pipelined nor a ROM-based evaluation logic implementation may suffice. In particular the application studied in Chapter 6 demands a high clock frequency of 225 MHz which neither pipelined nor ROM-based evaluation logic can provide. In such a case, one may leverage existing work on control generation. However, this may require non-trivial integration efforts, because the architectures in which the related works are used differ from the LAURA architecture. We present three alternative works that may be considered when further improving the LAURA evaluation logic components.

The CLooGVHDL tool generates a VHDL controller which traverses the points of a set of polytopes according to a predefined order [DBC+07]. The controller consists of a set of communicating automata that iterate over the dimensions of the polytope.

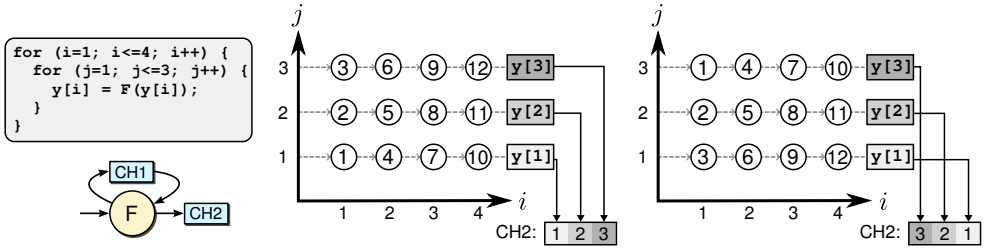
By placing registers between the automata, the maximum achievable clock frequency can be increased. Parallel execution of multiple instances of statements was left as future work. This would be of interest to us, since such parallel execution occurs in the LAURA architecture.

PARO attempts to reduce the resource cost of control logic by identifying counters and control signals that can be shared across different processors [DHRT07]. This approach was shown to lower resource cost particularly for partitioned applications, since the different partitions still have parts in common. However, the efficacy of this is limited for PPNs implemented using LAURA processors because of the globally asynchronous nature of the PPN model. That is, although two processes may share the same process domain and thus have similar control logic, they do not necessarily traverse their domains at the same pace.

Another alternative for the evaluation logic components of a LAURA processor is to implement them using existing HLS tools such as AutoESL [Xil11] or SynphonyC [Syn10]. This has the advantage that a target clock frequency can be specified. The HLS tool then produces a pipelined controller that is optimized for the specified clock frequency. However, we found that in practice the output of such tools have difficulties with the read and write units of a LAURA processor being decoupled [HK09]. For example, stalling the generated controllers on a blocking read condition was not fully supported at the time of our investigation. When such implementation problems have been resolved by HLS tool vendors, using an HLS tool to generate the evaluation logic might be the most favorable alternative solution.

3.6 Out-of-Order Communication

Ideally, a producer process produces tokens in the same order as the consumer process consumes them. Such *in-order* communication allows the channel from producer to consumer to be realized using a relatively inexpensive FIFO buffer. However, the PPNs of some applications do not exhibit solely in-order communication, as explained in Section 2.3.1. On some channels the order in which tokens are produced by the producer process may be different from the order in which tokens are consumed by the consumer process, and vice versa. Such communication is known as *out-of-order communication*. Out-of-order channels cannot be realized using FIFO buffers, because the token order needs to be taken into account to guarantee functional correctness. Instead, more sophisticated interconnects are required, such as *reordering buffers*. Reordering buffers store incoming tokens in order in a private memory and contain reordering logic which outputs the stored tokens in the order required by the consumer. Alternatively, circular buffers with overlapping windows



a) C program and PPN. b) Execution according to original program. c) Execution with inner loop reversed.

Figure 3.9: Two executions of a program with different communication behavior.

can realize out-of-order communication [BBS09]. This solution requires modifications to the producer and consumer process synchronization primitives. The impact on performance and resource cost of these modifications, and the performance and resource cost of the buffer itself is unclear, as no RTL implementation case study has been conducted yet.

In Figure 3.9, we show an example C program and two valid executions of this program. In the first execution shown in Figure 3.9b, we follow the execution order of the original program. That is, we first execute $(i, j) = (1, 1)$, followed by $(1, 2)$, etc. The relative order of iteration executions is illustrated by the number inside the points of Figure 3.9b. Only when $i = 4$, tokens are written to channel *CH2*. Channel *CH2* receives tokens in the order `y[1]`, `y[2]`, `y[3]`. Another valid execution in which the inner loop is traversed in the reverse direction is shown in Figure 3.9c. As a result, channel *CH2* receives tokens in the order `y[3]`, `y[2]`, `y[1]`, which is different from the order shown in Figure 3.9b. If we assume that *CH2*'s consumer process is not modified, the tokens would arrive in reverse order if *CH2* would be implemented using a FIFO buffer. To respect the correct token order, channel *CH2* has to be implemented using a reordering buffer.

Turjan et al. have proposed different realizations of reordering buffers, such as linear, pseudo-polynomial, and Content Addressable Memory (CAM) based implementations [TKD03]. The authors showed that these reordering buffer designs have a considerable negative impact on performance and resource usage. For example, read and write operations of a CAM implementation take four and two clock cycles [ZTKD02], respectively, while read and write operations on a regular FIFO take only one clock cycle.

To avoid counteracting the benefits of an application transformation because of possible reordering communication, we have developed a new reordering buffer [HK12]. The primary difference with previous work is that read and write operations now take

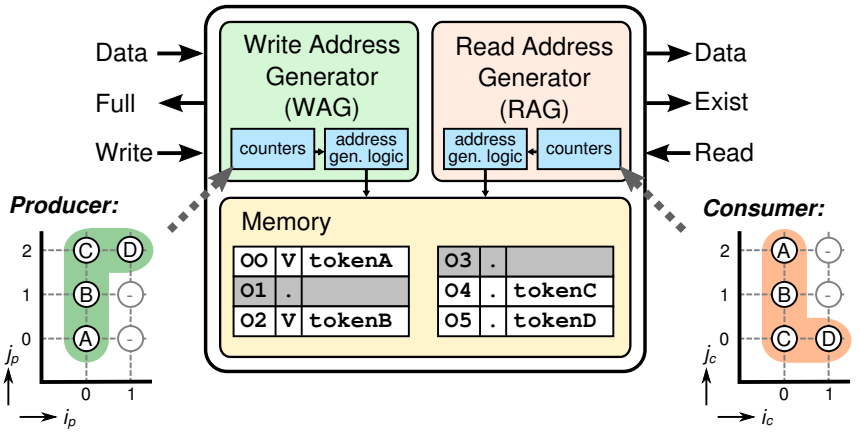


Figure 3.10: Reordering buffer.

only one clock cycle. This means that replacing a FIFO buffer with a reordering buffer increases resource usage, but does not introduce additional delay cycles.

Our reordering buffer is composed of a Write Address Generator (WAG), a Read Address Generator (RAG), and a private memory. The memory is dual-ported, with one port being addressed by the WAG and the other port being addressed by the RAG. The WAG and RAG both contain a set of counters which iterate through domains associated to the channel. These counters are used by the address generation logic to compute the next write and read addresses. To avoid delay cycles, the counters and address generation logic are implemented in a pipeline fashion. To minimize the latency of the address generation logic, we employ a linear addressing scheme. This addressing scheme is based on conventional linearization of an n-dimensional array into a 1-dimensional array. As such, the resulting address expressions are linear polynomials that can be realized efficiently in hardware.

The interface of the reordering buffer resembles a point-to-point FIFO buffer interface. This allows straightforward integration of reordering buffers in ESPAM-generated PPN implementations. That is, when a transformation introduces out-of-order communication, we do not have to modify the interfaces of the processes involved in the out-of-order communication. The interface is depicted in Figure 3.10. The outgoing slave interface exposes an output data bus, an exist signal to indicate if a token is available, and a read signal to acknowledge a read operation. The incoming master interface exposes an input data bus, a full signal to block write operations when the buffer is not ready to accept them, and a write signal to acknowledge a write operation.

We illustrate the memory organization of our reordering buffer at the bottom part of Figure 3.10. In the bottom left, we show a producer domain consisting of four points $(0,0)$, $(0,1)$, $(0,2)$, and $(1,2)$. The producer produces four tokens in the order A, B, C, D. We store these tokens according to a linear addressing scheme at address

$$\mathit{wAddr}(i_p, j_p) = i_p + 2 \cdot j_p. \tag{3.4}$$

The slot for each token is shown in the memory of Figure 3.10. For example, token C is produced in iteration $(0,2)$ and is therefore stored at address `04`. Because of the linear addressing scheme, some addresses may remain unused for non-rectangular domains. In our example, this occurs for addresses `01` and `03`. The consumer domain shown on the bottom right consumes the four tokens in the order C, D, B, A. To retrieve these tokens in the correct order from the memory, we compute

$$\mathit{rAddr}(i_c, j_c) = \mathit{wAddr}(M_{p \to c}(i_c, j_c)) \tag{3.5}$$

for each point in the consumer domain. That is, we first apply the channel relation $M_{p \to c}$ as found by the PN compiler. This gives the point (i_p, j_p) in the producer domain that corresponds to the point (i_c, j_c) in the consumer domain. We then compute $\mathit{wAddr}(i_p, j_p)$ to obtain the address from which the token should be read. For the example of Figure 3.10, PN finds the channel relation

$$M_{p \to c}(i_c, j_c) = \begin{bmatrix} i_p \\ 2 - j_p \end{bmatrix}. \tag{3.6}$$

Therefore, the read address function becomes

$$\mathit{rAddr}(i_c, j_c) = i_c + 2 \cdot (2 - j_c). \tag{3.7}$$

For token C, which is consumed in iteration $(0,0)$, the *rAddr* function yields address `04` which is the same address that was computed by the WAG. However, a token may not have been written by the producer yet. For example, token C may not be available yet at address `04`. Therefore, we introduce an additional valid bit for each memory location. The valid bit is set once a token has been written to its address. To comply with the blocking read semantics of the PPN model, the RAG blocks until the token corresponding to the current consumer iteration is written. In the memory of Figure 3.10, tokens A and B have been written, as indicated by the "v"s, whereas tokens C and D have not been written yet, as indicated by the "."s.

3.7 Conclusion and Summary

To realize the complete forward synthesis flow from a C specification to an FPGA implementation (cf. Figure 1.3), we have presented four extensions to the LAURA methodology in this chapter. These extensions include a more flexible characterization of IP core performance and resource cost aspects; support for novel optimizations of the PNGEN tool flow; architectural optimizations to improve the maximum clock frequency and handle complex iteration domains; and a novel reordering buffer implementation that has a lower performance penalty compared to previous reordering buffer implementations. The extensions enable the Daedalus tool flow to support transformations and cope with industrially relevant applications, as we show in the next chapters.