Automated Synthesis of Hardware Process Networks from Sequential C Code

M.Sc. thesis of

Sven van Haastregt

August 25, 2008

LIACS, Leiden University

Supervisors: Dr. ir. A.C.J. Kienhuis
Prof. dr. E.F. Deprettere
Student ID: 0308862

Abstract

In this thesis, we present a methodology for the automated generation of complete hardware implementations from C input specifications. Our methodology is based on the Kahn Process Network (KPN) Model of Computation (MoC). By leveraging previous research, we can automatically obtain a parallel KPN representation of sequential C input code. Next, we synthesize RTL implementations of the KPN nodes, by making use of PICO, a high level synthesis tool, to automatically obtain an RTL implementation of the node functionality. An important benefit of our approach is that source code annotations that indicate parallelism or influence low-level implementation decisions are not required for efficient results. This allows for an efficient software engineering design flow to program FPGAs. We show that by applying transformations to KPNs, one can obtain different application instances with improved throughput of the resulting hardware implementations. We also show that the distributed memory model inherent to the KPN MoC suits well to customizable hardware target platforms like FPGAs.

Acknowledgements

I would like to thank everyone who has contributed to this thesis. Special thanks go to the people of the Leiden Embedded Research Center (LERC) of the Leiden Institute of Advanced Computer Science (LIACS), in particular my supervisor dr. ir. Bart Kienhuis, and prof. dr. ir. Ed Deprettere, Sjoerd Meijer, Hristo Nikolov and Todor Stefanov. I would also like to thank dr. Vinod Kathail, Pradeep Thiruchelvam and Fernando Martinez at Synfora, Inc., for offering me the opportunity to use the PICO tool and getting me familiar with it. Also, I would like to thank Kees Vissers and Stephen Neuendorffer at Xilinx, Inc. for their technical support and the useful discussions we had at Xilinx Headquarters in San Jose (CA). Finally, I would like to thank my parents for providing a great living and working environment, thereby enabling me to fully concentrate on this work.

Contents

Chapter 1

Introduction

Currently, electronic devices are used on a very large scale in many different fields. Examples include consumer electronics, like cell phones or DVD players for example. Also the industry and scientific fields depend heavily on modern electronics: in virtually all industrial and scientific environments where automation is involved, microchips play an important role.

In almost all of these areas, the demand for compute power is continuously increasing. There are several reasons for this. Users require more functionality and applications get more complex. Advances in other fields increase this demand even more. For example, think of new high resolution Magnetic Resonance Imaging (MRI) scanners, telescope arrays, exploration geophysics equipment and meteorological systems. New generations of these devices produce much more raw data than their predecessors. These vast amounts of raw data must be processed into some human-intelligible form, which is a compute intensive job.

In many of the aforementioned applications, time plays a crucial role. Data processing should take place within a reasonable amount of time, such that the field expert can obtain the required information on time and respond accordingly if necessary. This means the system should have a high *throughput* and must meet performance targets. For a typical compute intensive job, a microprocessor based solution will not be satisfactory. This is partly due to its lower execution speeds, in exchange for ease of programming, but also due to its sequential nature. Particularly when a great portion of the compute job can be performed in parallel, as is typically the case with data processing applications, a single thread of execution will still yield a very low throughput.

To achieve a high throughput, one should switch to a platform that offers possibilities to ex-

ploit the available parallelism. This could be a multi-core microprocessor system, like the Cell Broadband Engine [1], or a grid of multiple compute nodes. However, when *streaming data* applications are considered, communicating data to the various cores or nodes is a critical factor. A core or node that does not receive data on time will stall, leading to a suboptimal utilization of resources and decreased performance. Hence, expensive high-bandwidth interconnections are required to make sure the cores or nodes can operate without having to wait for data communication. Unfortunately, this is not always possible. Because clock frequencies of modern processing units have increased much faster than the throughput of the interconnections between them, stalls are sometimes inevitable. This prevents the system from running at its maximal speed, reducing throughput.

By going to a smaller implementation scale, communication constructs are typically less complex. Particularly if all data communication takes places on the same chip in a neighbour-to-neighbour fashion for example, fast communication is easily realized using plain wires and logic gates. Such implementations are obtained by creating a low level hardware description of the desired functionality. This description can then be used to create an *Application Specific Integrated Circuit* (ASIC) or to program reconfigurable hardware like a *Field Programmable Gate Array* (FPGA). This hardware based approach offers several other advantages. A dedicated hardware implementation does not come with the overhead of an implementation on a generic platform. For example, only the functional units that are required are included in the design, data bus widths can be chosen as necessary and control is significantly less complex than that of a microprocessor. The hardware implementation can be built to precisely fit the needs of the application. These differences lead to increased performance and a reduced number of gates needed for a physical hardware product. However, implementing algorithms in hardware is not a trivial task. The way of programming or configuring such platforms does not match a typical algorithm specification written in a high level language like C for example.

1.1 Problem Definition

Due to the mismatch of the algorithm specification and the target platform, most of the hardware implementations of algorithms are currently developed manually. This is a complex, time-consuming and error-prone process. Extensive knowledge about platform characteristics is required to get to an efficient implementation that satisfies performance and cost constraints. Usually, an algorithm designer or software engineer does not possess this knowledge. Thus, the hard-

ware platforms often remain out of reach for them.

Maintaining hardware designs is another issue. Because of the low level of abstraction of hardware designs, which are usually written at the *Register Transfer Level* (*RTL*) in hardware design languages (*HDLs*) like VHDL or Verilog, the hardware design quickly becomes complex. When the requirements of the application change, updating the hardware design accordingly is again a difficult and time-consuming task.

From a software engineering point of view it would be interesting if a high-level algorithm specification could be directly translated into a hardware implementation. From a hardware engineering point of view it would be interesting to specify a system at a high level of abstraction, enabling easy maintenance, while automated synthesis of this high level design still results in a cost and performance equal or close to a manually constructed low level design.

Unfortunately, most high level languages are based on a single sequential thread of control while, on the other hand, custom hardware offers many opportunities to exploit parallelism. This means we need to automatically search for parallelism in the sequential input specification and subsequently exploit the obtained parallelism in an efficient way by taking advantage of the flexibility of custom hardware. This is not straightforward and is still subject of ongoing research [2, 3, 4, 5, 6]. To complicate matters, the established high level programming languages like C or C++ are tightly coupled to the von Neumann architecture [7]. This perfectly matches the shared memory architecture of generic microprocessor systems, but mapping such code to custom hardware poses an additional challenge, because a distributed memory layout is in general more efficient on such a platform.

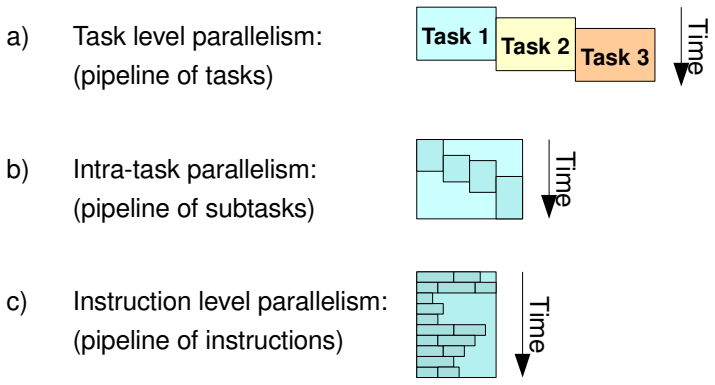


Figure 1.1: Different levels of parallelism.

In order to attack the problem sketched above, it is important to realize that different levels of parallelism are available to be exploited. In Figure 1.1a, *task level parallelism* is depicted, where multiple instances of the same task operating on different input data run in parallel. In Figure 1.1b, *intra-task parallelism* is depicted, where parallelism inside a task is exploited by decomposing the task into different subtasks and pipelining the execution of those subtasks. In Figure 1.1c, *instruction level parallelism* is depicted, where the low level operations of each subtask are scheduled in parallel.

Taking these observations into account, we want to address the following problem. How can we automatically derive efficient FPGA implementations from a sequential C specification, taking into account the fact that we can exploit various levels of parallelism and the fact that we can adapt the platform to precisely fit our needs?

1.2 Solution Approach

To get from sequential code to an efficient hardware realization, we have developed the approach that is depicted in Figure 1.2. As a first step, depicted by the "Compiler" block, we extract coarse-grained parallelism from the sequential input specification. This results in a network of multiple smaller units of execution which we call *processes*, which allows us to easily exploit task level parallelism as depicted in Figure 1.1a. Next, we synthesize a hardware implementation for each of these processes and connect the RTL cores according to the network topology. This is depicted by the "Synthesis" block of Figure 1.2. During this phase, finer-grained levels of parallelism are extracted and exploited, which are shown in Figures 1.1b and 1.1c.

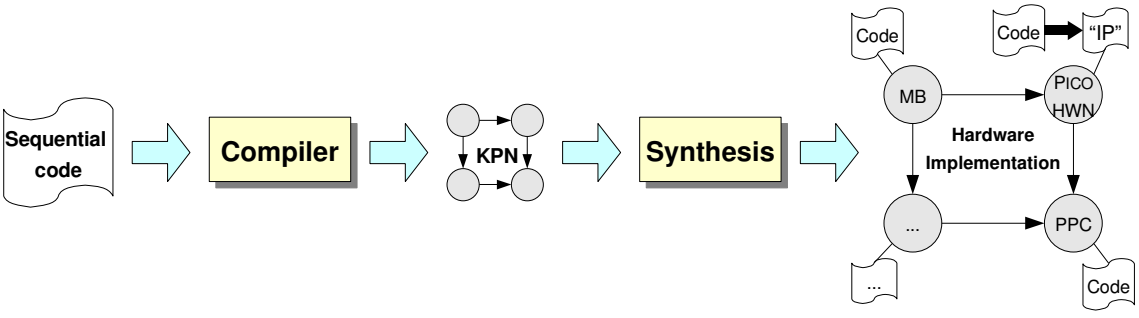


Figure 1.2: A high level overview of our solution.

In order to describe the network of processes and their behaviour, we use the *Kahn Process Network* (*KPN*) Model of Computation (MoC) [8], which has proven to be appropriate in similar cases [9, 10]. This model is discussed in Section 2.1. An advantage of this MoC is that high level transformations [11] can be applied to a KPN, like unrolling and skewing, offering the possibility to explore various alternative implementations of the same application.

To partition the sequential C input specification into a parallel KPN representation, we use the KPNGEN tool. This tool is discussed in Section 2.2. Next, we feed the obtained KPN to ESPAM-PICO, which is an extended version of ESPAM. Based on a platform and mapping specification, ESPAM generates a synthesizable hardware implementation of the KPN. ESPAM is discussed in Section 2.3. Both KPNGEN and ESPAM are part of the Daedalus framework [12, 13], a collection of open-source tools intended for system-level architectural exploration and high-level synthesis.

ESPAM relies on a library of IP cores in order to deliver a complete hardware implementation of a given KPN. Our extension to ESPAM, which we describe in Chapter 4, does not require this IP core library anymore. Instead, ESPAM-PICO invokes another tool which generates custom IP cores from C code that implement the functional behaviour of the various nodes of the KPN. This is depicted in the upper right corner of Figure 1.2. The functional behaviour is taken from the original input specification, which is written in a subset of C.

To generate IP cores from C code, various tools are available, both commercial and non-commercial. These tools are capable of generating a hardware implementation from a high(er) level language input specification. See Section 1.3 for an overview of such tools. In general, these tools accept a subset of the C language and produce an RTL implementation. An example of such a tool is Synfora PICO [2], a commercial product of Synfora, Inc. This tool exploits parallelism at the various levels shown in Figure 1.1 and applies sophisticated scheduling techniques to obtain an efficient RTL implementation in terms of area and performance. PICO is invoked by our ESPAM-PICO tool to generate the custom IP cores. PICO offers streaming interfaces for the generated cores. These streaming interfaces have FIFO semantics, which makes the cores fit well in our network of communicating processes.

1.3 Related Work

In this section, we give an overview of work related to automated hardware generation from a high-level input specification. Many tools and techniques have been developed over the years to convert a specification in a high level language into a (synthesizable) representation closer to

the hardware level. These tools typically restrict the set of accepted input specifications to a class of specifications for which the tool can derive efficient implementations. Moreover, special annotations or code restructuring is often required to obtain efficient results.

The Handel-C language [14] is a small subset of C, extended with some constructs to influence the efficiency of generated hardware. The SpecC language [15, 16] is intended for specification and design of embedded systems, including hardware and software portions. The Alpha language [17, 18] is a functional language intended for systolic array synthesis research. The language is based on systems of affine recurrence equations, which makes it quite a different language compared to the other (C based) high level languages discussed in this section. Using a series of transformations, an Alpha program can be converted into a netlist, as described in [19]. The ROCCC compiler [3] generates VHDL from a subset of the C language. ROCCC targets applications that have a high computational density and a low control density. It employs a sophisticated sliding window approach for off-chip memory accesses, although this only improves performance for particular (consecutive) memory access patterns. Our distributed memory based approach can be applied to more irregular access patterns as well. SPARK [20, 21] accepts a subset of ANSI-C as input, applies optimization and scheduling techniques and generates VHDL. However, it does not support multi-dimension array accesses, which are typical in image processing applications for example. Trident [22] is a C-to-VHDL compiler that particularly focuses on floating point arithmetic. SA-C (Single Assignment C) [23, 24] is a single assignment variant of a subset of the C language. After various optimizations, VHDL components are generated from data flow graph representations. The SA-C language aims at image processing applications in particular. Streams-C [25, 26] is an extension to C by means of source code annotations and library functions. This way, a Communicating Sequential Processes (CSP) parallel programming model based on C-like syntax is offered. The project supports both VHDL generation and functional simulation of applications written in the Streams-C language. Impulse C [27] is a commercial tool that is similar to Streams-C, in terms of programming model and operation. DWARV [6] is a C-to-VHDL tool targeted towards the MOLEN [28] polymorphic processor paradigm. No C syntax extensions are used, but pragma annotations are necessary.

The PARO [29, 30] design flow accepts sequential nested loop programs written in a subset of C. The loop nests are parallelized, loop transformations are applied and design space exploration is performed. Finally, VHDL code representing an array of processing elements is generated, including communication and control components.

The Catapult Synthesis [4] tool suite accepts unannotated ANSI C/C++ as input and generates an

RTL implementation. However, the C code has to be written according to the Catapult-C coding guidelines in order to obtain efficient results.

'Machines' [31] is a programming model that requires the user to specify medium-grained parallelism. The compiler takes object oriented C++ code that is specified according to the programming model and translates this into Predicated Static Single Assignment code (PSSA). Optimizations are applied and finally a bitstream is generated that can directly be used on an FPGA.

Disydent/UGH [32, 33] is a set of tools that translates a KPN-based program specification into synthesizable VHDL descriptions. The tool set expects the user to provide a C program, partitioned using POSIX threads, and a high level description of the KPN. The UGH tool then generates the hardware implementation. As the name UGH (User Guided High level synthesis) suggests, the compilation process depends heavily on decisions of the user. Hence, the designer should have thorough knowledge of the underlying techniques in order to obtain efficient hardware implementations.

CLooGVHDL [34] is an extension to CLooG [35], a tool that generates code for traversing the integral points of parameterized polyhedra. Currently, the set of acceptable input programs is restricted because of the use of the polyhedral model. CLooGVHDL first calculates reuse distances of the memory references in the input program. With this information, a set of loop transformations is determined that improve temporal data locality. The transformed polyhedral representation of the program is then converted into hardware. This hardware implementation consists of two entities: the implementation of the statements, typically assignment statements, and the controller that updates the iterators and triggers the statements at the right moment. At this point, a purely sequential hardware implementation of the program is realized. Now, dependence analysis can indicate which loops and/or statements can be executed in parallel. By duplicating parts of the controller, different loops can be executed in parallel. By duplicating the implementation of a statement, multiple instances of this statement can be executed in parallel. Clearly, this involves a trade-off between execution speed and chip area. The final output is presented in the form of VHDL code. Unfortunately, this tool does not produce fully functional implementations, as the VHDL implementations of non-control statements are not generated automatically.

A different approach is to create new hardware design languages that try to combine constructs from high level languages (like C or C++) with constructs from hardware description languages. This provides a very versatile language, but there are some disadvantages: existing code needs to be translated into the new language and the programmer often needs to indicate parallelism to some extent.

HardwareC [36] is such a "new" hardware description language that has a C-like syntax. The language is extended with concepts like concurrent processes, message passing, timing constraints and resource constraints. The Olympus Synthesis System [37] takes input specifications written in HardwareC and offers chip level synthesis or simulation of designs.
In [38], Superlog is proposed. This language tries to combine the hardware description features of Verilog and the general purpose programming constructs of C, like structures and pointers, into a single language.
SystemC [39] is a C++-based hardware/system description language aimed at system design and verification. SystemC designs can be simulated and synthesized to an RTL description or netlist, although verification currently seems to be the most widespread use of the language. In [40], a method for SystemC code generation from Unified Modelling Language (UML) diagrams is described.

It is also possible to develop a new platform and design a high level programming language specific to that platform. This provides a fast way to accelerate an application using hardware and prevents the user from having to deal with low level constructs. However, applications need to be rewritten in the appropriate language and designs for such a platform are not directly portable to other platforms.
The Mitrion platform [41] is based on such an approach. Applications have to be written in the Mitrion-C language. A compiler then instantiates a Mitrion Virtual Processor and adapts it to the needs of the application. This Mitrion Virtual Processor is a soft-core processor that can be instantiated on an FPGA for example.
The Carte Programming Environment [42, 43] of SRC Computers accepts plain, unannotated Fortran or C code and generates a so called "unified executable" for use with a MAP Processor. This is a reconfigurable platform that has also been developed by SRC Computers, Inc. Although the input code can be kept free of tool-specific annotations, the result is subject to the constraints of the MAP processor. This limits flexibility and scalability.

1.4 Thesis Organization

The remainder of this thesis is structured as follows: In Chapter 2, we explain the model of computation being used, as well as some existing tools that are used in our approach. In Chapter 3, we propose two new hardware node models and in Chapter 4, we show how we can automatically generate hardware implementations of an application using these models. In Chapter 5, we explain

the experiments that we have conducted with our approach and show the obtained results. Finally, in Chapter 6, we mention possible future work and in Chapter 7, we summarize our work and our findings.

Chapter 2

Background

In this chapter, we discuss the KPN model of computation and we give an overview of three existing tools that we use in our approach.

2.1 The Kahn Process Network Model of Computation

A Kahn Process Network, or KPN, is defined as a directed graph $G = (V, E)$, where $V = \{p_1, \ldots, p_N\}$ is a set of concurrently executing processes, represented by the vertices or nodes of the graph, and $E = \{e_1, \ldots, e_M\}$ is a set of *FIFO* (First In, First Out) channels, represented by the edges of the graph. The KPN model of computation is deterministic: the result(s) of the computation, that is, the data transferred on the FIFO channels, will be the same for all possible firing sequences of the network.

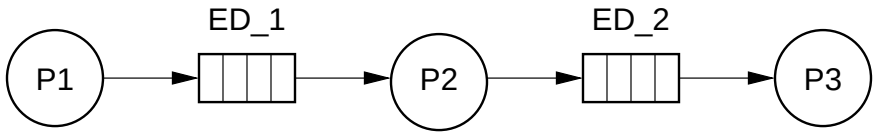


Figure 2.1: An example of a Kahn Process Network, consisting of three nodes and two FIFO channels.

In Figure 2.1, an example of a KPN is shown, which consists of a producer node *P1*, a transform node *P2* and a consumer node *P3*. Process *P1* can send data to process *P2* via FIFO channel

ED_1 and process *P2* can send data to process *P3* via FIFO channel *ED_2*.

Each of the processes in a KPN is sequential and follows a fixed internal execution schedule. However, there is no global execution schedule. Communication between processes is accomplished by means of unbounded FIFO channels. Each node has zero or more incoming FIFO channels, and zero or more outgoing FIFO channels. The incoming channels are connected to the *input ports* of a node; the outgoing channels are connected to the *output ports* of a node. A process can send *tokens* to its outgoing channels and receive tokens from its incoming channels by means of atomic write and read operations. The write operation is non-blocking, meaning that it always succeeds without delay. The read operation is blocking, meaning that execution of the entire process halts if the channel on which the read operation was performed is empty. Once data becomes available again, execution is resumed.

The unboundedness of the FIFO channels does not allow an implementation on a platform with a finite amount of memory. Hence, for a real implementation each FIFO i is bounded by some value S_i and the non-blocking write operation is changed into a blocking write operation that blocks when the channel written to is full. However, when one or more buffer sizes are chosen too small, an *artificial deadlock* may occur [44]. In such a case, none of the processes can make progress anymore because they are directly or indirectly waiting on one or more processes that are blocking on a write operation. In the remainder of this thesis, we assume that buffer sizes are chosen large enough to prevent such artificial deadlocks.

2.1.1 Transformations

A strong point of the KPN model is that we can explore alternative instances of an application by applying high level transformations to the application source code. By translating the transformed source code into a KPN, this new KPN exhibits the characteristics that were intended by the transformation. Transforming the application source code can be automated, as illustrated in [11, 45]. In this thesis, we consider two transformations, namely *unrolling* and *skewing*.

Unrolling

The unrolling or “process splitting” transformation is applied to one node of a KPN at a time. This node is replaced by U adjusted copies of the node. Here, U is called the *unroll factor*. The functionality of these new nodes is modified such that each node performs a different portion of the computational workload of the original node. This may be done by adding if-statements to

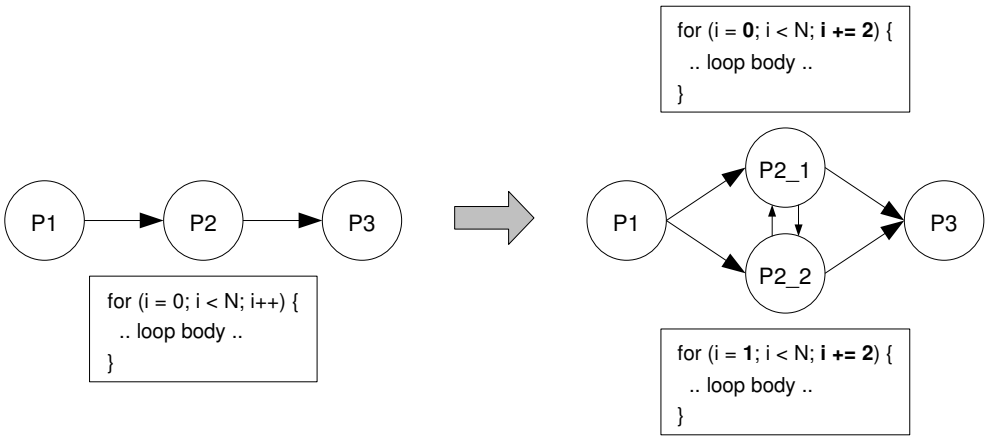


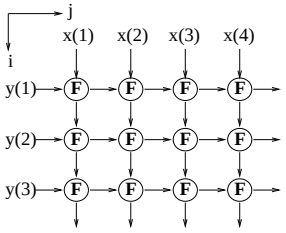
Figure 2.2: An example of an unrolling transformation with a factor of 2 applied to node *P2*.

the process code, or by directly changing the loop bounds and step sizes, as we did in Figure 2.2. In the ideal case, the new nodes can now operate in parallel and finish the entire computation in less time than the unmodified network would need. The unrolling transformation leads to an increased number of nodes and edges in the network, but the functionality remains the same. The predecessor and successor nodes of the transformed node also need to be adapted, because the predecessor nodes now need to select the right destination node for each token they send and the successor nodes need to collect tokens from the right input channels. This can be seen in Figure 2.2, where the amount of outgoing channels of node *P1* and the amount of incoming channels of node *P3* differ for the original and the transformed network. Additional edges between the unrolled nodes may be required, depending on the presence of loop-carried dependencies in the original process code.

Skewing

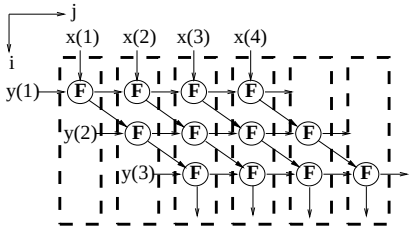
The skewing transformation can be used to make potential parallelism of the input application explicit. This is done by adjusting the loop bounds and (array) variable indices of the code that belongs to a node. After applying the skewing transformation, iterations that could not run in parallel in the original application may now execute in parallel. The skewing transformation might lead to improved pipeline efficiency of the operations inside a node because there are less data dependencies that could cause stalls. By combining this with the unrolling transformation, even

```
for j = 1:1:4,
   for i = 1:1:3,
      [y(i), x(j)] = F(y(i), x(j));
   end
end
```



a) Application program and its dependence graph.

```
for j =  2:1:4+3,
   for i = max(1,j−4):1:min(j−1,3),
      [y(i), x( j−i)] = F(y(i), x( j−i));
   end
end
```



b) Same program with its i−loop skewed.

Figure 2.3: An example of a skewing transformation.

shorter execution times can be obtained.

In Figure 2.3, an example of a skewing transformation is shown. In Figure 2.3a, the original input is shown on the left, and the corresponding Dependence Graph (DG) is shown on the right. In this DG, the nodes represent the workload of the iterations, that is, the function calls, and the edges represent the data dependencies between the function calls of the iterations. In Figure 2.3b, a version of the application with its i-loop skewed is shown. Now, the DG graph explicitly indicates which iterations can be executed in parallel due to the absence of data dependencies between their nodes. Such sets of iterations are placed inside a dashed box.

2.2 KPNGen

KPNGEN [46] is a chain of tools that takes a C or C++ file as input and generates a KPN description, as depicted in Figure 2.4. Currently, the input to KPNGEN is restricted to Static Affine Nested Loop Programs (SANLPs). This means control must be static and expressions in loop bounds, array accesses and if-statements must be affine combinations of iterators and parameters.

Figure 2.4: The KPNGEN tool flow.

First, the input source code is converted into *Single Assignment Code* (SAC) which resembles the *Dependence Graph* (DG) of the original program. This SAC is then converted into a Polyhedral Reduced Dependence Graph (PRDG). This is a compact mathematical representation of the DG, based on the polyhedral model. Finally, the PRDG is converted into a KPN. The nodes of the PRDG correspond to the processes of the KPN. The data dependencies of the PRDG correspond to the communication channels of the KPN. For each channel, a buffer size is computed based on a deadlock-free schedule. Note that this particular deadlock-free schedule may not be optimal, and thus the computed buffer sizes may not be valid for the optimal schedule. However, a valid schedule exists for the computed buffer sizes.

2.2.1 Input

We illustrate the flow of KPNGEN by means of an example. In Figure 2.5, an example of a SANLP written in the C language is shown.
On lines 3 & 4, the parameter N is defined. Starting at line 6, the top level procedure that is to be converted into a KPN is declared. Lines 10–12 initialize array a by means of function calls to the `source` function. Here, the "addressOf" operator (`&`) is used in the argument expression to allow element `a[i]` to be written. At lines 14–16, the `transform` function is called for each element of array a. This function writes its results to array b. Finally, at lines 18–20, the elements of array b are consumed by the `sink` function.

2.2.2 Tool invocation

The Daedalus framework provides a script which invokes the appropriate tools in the correct order. The `pn` subtool is controlled using the `PN_OPTIONS` environment variable. This allows one to influence the characteristics of the network. For example, the `--no-reuse` option generates

```
1   #include "funcs.h"           // Contains function prototypes
2
3   #define N 16
4   #pragma parameter N 16 100
5
6   int main() {
7     int i;
8     int a[N+1], b[N+1];
9
10    for (i = 1; i <= N; i++) {
11      source( &(a[i]) );
12    }
13
14    for (i = 1; i <= N; i++) {
15      transform( a[i], &(b[i]) );
16    }
17
18    for (i = 1; i <= N; i++) {
19      sink( b[i] );
20    }
21
22    return 0;
23  }
```

Figure 2.5: Example input to KPNGEN.

a network that does not contain reuse channels. Enabling this option generally leads to a lower number of channels, but might lead to an increased number of the more expensive reordering channels.

2.2.3 Output

After invocation of the KPNGEN script, a KPN of the input specification is produced. The KPN is offered in both an YAML and XML format. Because we pass the KPNGEN output on to ESPAM, we are only interested in the XML output. In the following paragraphs, we highlight the most important features of the XML representation of the generated KPN for the input program of Figure 2.5.

The XML output file begins with the following lines, defining the document type and the start of the Approximated Dependence Graph (ADG):

```
1 <?xml version="1.0"?>
2 <!DOCTYPE sadg PUBLIC "-//LIACS//DTD ESPAM 1//EN"
3   "http://www.liacs.nl/~cserc/dtd/espam_1.dtd">
4 <sadg>
5   <adg name="example" levelUpNode="">
```

Next, for each node of the KPN, a `node` element is given. Below is the corresponding declaration of node `ND_1`, which corresponds to the `trans` function call in the example.

```
6     <node name="ND_1" levelUpNode="">
7       <inport name="ND_1IP_ED_0_0_V_0" node="ND_1" edge="ED_0">
8         <bindvariable name="in_0" dataType="int"/>
9         <domain type="LBS">
10          <linearbound index="c0" staticControl="" dynamicControl="" parameter="">
11            <constraint matrix="[1, 1, -1; 1, -1, 16]"/>
12          </linearbound>
13        </domain>
14      </inport>
```

The `inport` element defines an input port of the current node. This input port operation corresponds to the read operation of array element `a[i]` in line 15 of Figure 2.5. The variable that is bound to this input port is `in_0`, as defined by the `bindvariable` tag. The `domain` element in lines 9–13 contains information about the iteration space of the input port. The constraint matrix that is given in line 11 looks as follows:

$$M_c = \begin{bmatrix} 1 & 1 & -1 \\ 1 & -1 & 16 \end{bmatrix}$$

Each row of M_c represents a constraint. If the first element of a row equals zero then the constraint is an equality ($= 0$); if the first element equals one then the constraint is an inequality (≥ 0). The next columns of M_c contain the coefficients of the (control) variables. In this example there is only one variable, namely the c_0 index variable which is declared in line 10 of the XML. The last column contains the constant of the constraint. By interpreting M_c accordingly, we get the following constraints on the iteration space of the current input port statement:

$$I_{ND\_1IP\_0} = \left\{ c_0 \mid \begin{array}{r} c_0 - 1 \geq 0 \\ -c_0 + 16 \geq 0 \end{array} \right\} = \{c_0 \mid 1 \leq c_0 \leq 16\}$$

Note that this set of integers exactly matches the iteration space of the second for-loop in Figure 2.5. Similarly, an output port is defined, corresponding to the write operation to array element `b[i]`, in line 15 of Figure 2.5. The iteration space of this output port is equal to that of the input port discussed above; hence we have omitted the `domain` element in the following fragment:

```
15        <outport name="ND_1OP_ED_1_0_V_1" node="ND_1" edge="ED_1">
16          <bindvariable name="out_1" dataType="int"/>
17          ...
18        </outport>
```

Next, the parameter signature of the function associated to this node is described:

```
19        <function name="trans">
20          <inargument name="in_0" dataType="int"/>
21          <outargument name="out_1" dataType="int"/>
22        </function>
23        ...
24      </node>
```

Finally, a `domain` element describing the iteration space of the node is given. Again, this one is omitted in the fragment above because it is equal to the other `domain` elements. For the `source` and `sink` function calls, nodes `ND_0` and `ND_2`, respectively, are created. The XML data describing these nodes is similar to the description of `ND_1`, with some obvious differences: `ND_0` does not possess any `inport` elements, because there is no valid data being read by the `source` function call. Likewise, `ND_2` does not possess any `outport` elements, because the `sink` function does not produce any data.

After the list of nodes, the edges of the KPN are listed. For each edge, an `edge` element is given. Our example KPN contains two edges. The fragment below shows how the type, size and connections of these edges are described. Both edges are of the FIFO type, with buffer sizes of one. The `fromPort` and `toPort` attributes connect the edge to the specified output and input port, respectively. The `fromNode` and `toNode` attributes show from which node the directed edge is coming and to which node it is connected.

```
25      <edge name="ED_0" fromPort="ND_0OP_ED_0_0_V_0" fromNode="ND_0"
26                        toPort="ND_1IP_ED_0_0_V_0" toNode="ND_1" size="1">
27        <linearization type="fifo"/>
28        <mapping matrix="[1, 0, 0; 0, 0, 0]"/>
29      </edge>
30      <edge name="ED_1" fromPort="ND_1OP_ED_1_0_V_1" fromNode="ND_1"
31                        toPort="ND_2IP_ED_1_0_V_0" toNode="ND_2" size="1">
32        <linearization type="fifo"/>
33        <mapping matrix="[1, 0, 0; 0, 0, 1]"/>
34      </edge>
35    </adg>
```

Also, a graph in the Graphviz DOT format [47] is produced. This provides a visual representation of the network topology to the user. The graph produced for the example discussed in this section

is shown in Figure 2.6. In this figure, the node labels show the function name of the corresponding KPN process. The edges are labelled with the name of the variable in the original input associated to the data dependence, followed by the recommended minimum buffer size.

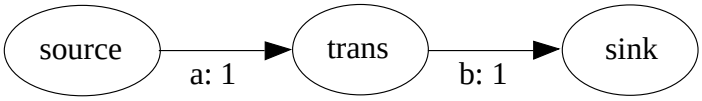


Figure 2.6: Automatically generated graph for the example input.

2.3 ESPAM

ESPAM (Embedded System-level Platform synthesis and Application Mapping) [48] is a tool intended for automated multiprocessor system design and implementation. The design flow is de-

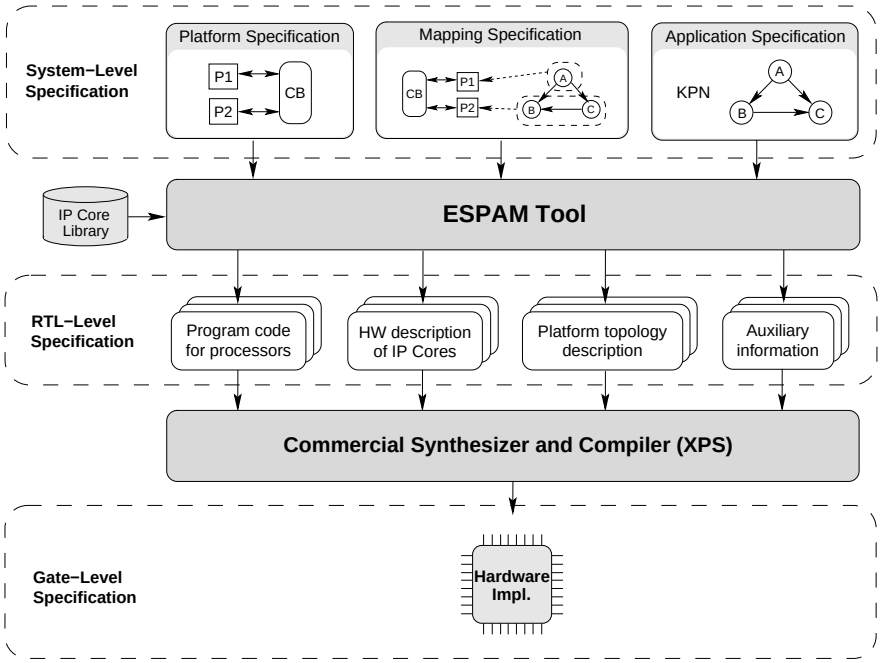


Figure 2.7: The ESPAM design flow.

picted in Figure 2.7. Starting from a high level system specification, the tool synthesizes and programs a multiprocessor system. The current version available at the time of writing targets Xilinx Virtex-II Pro FPGAs, by producing a Xilinx Platform Studio (XPS) project. It relies on the XPS tool to generate the final bitstream with which an FPGA can be configured. It should be noted that the XPS project is generated by the ESPAM back-end. In order to target a different platform, only the back-end has to be adapted.

2.3.1 Input

ESPAM requires three different XML files as input. First, an *application specification* is needed. The application is specified as a KPN in the format already described in Section 2.2.3. The output of the KPNGEN tool can be directly passed on to ESPAM as an application specification. Next, a *platform specification* is needed, which contains information about the processors and peripherals in a system and the interconnections between them. Finally, a *mapping specification* is needed, which maps the different processes of the application onto the processors of the system. The platform and mapping specifications are discussed below by means of an example that builds further upon the example of Section 2.2.1.

Platform Specification

In the platform specification, the various components of the system are specified. For an extensive discussion of the platform model used by ESPAM, we refer to [48]. In this section we only highlight the elements that are relevant to the remainder of this thesis. In Figure 2.8, an example platform specification is given.

A platform is composed of different components. The processing components are called *Processors*. In Figure 2.8, these are found on lines 2–10, identified by the `processor` XML elements. Each processor should be given a unique name such that it can be referred to. Next, the type of a processor should be provided. Currently, three different processor types are supported:

- **MB**: A 32-bit Xilinx MicroBlaze processor. This is a "soft processor core", that is synthesized out of the regular configurable logic of an FPGA. It allows some features, e.g., the presence of a hardware multiplier, to be configured by the user, thereby offering the option to reduce slice utilization in exchange for lower performance. ESPAM generates C code for the MicroBlaze processors, which is compiled using a C compiler during the final implementation phase.

```
1  <platform name="examplePlatform">
2    <processor name="MB_0" type="MB" data_memory="16384" program_memory="16384">
3    </processor>
4
5    <processor name="MB_1" type="MB" data_memory="16384" program_memory="16384">
6    </processor>
7
8    <processor name="MB_2" type="MB" data_memory="16384" program_memory="16384">
9      <port name="OPB_2" type="OPBPort"/>
10   </processor>
11
12   <peripheral name="UART_1" type="UART" size="256">
13     <port name="IO_1" type="OPBPort"/>
14   </peripheral>
15
16   <link name="mb_opb_2">
17      <resource name="MB_2" port="OPB_2"/>
18      <resource name="UART_1" port="IO_1"/>
19   </link>
20 </platform>
```

Figure 2.8: An example platform specification for the ESPAM tool.

- **PPC**: A 32-bit PowerPC processor. A Xilinx Virtex-II Pro FPGA provides up to two integrated PowerPC 405 cores. Again, ESPAM generates C code for this processor type which is compiled during the final implementation phase.

- **CompaanHWNode**: A processor similar to a node generated by the Compaan/ Laura chain [5]. ESPAM generates all necessary control logic for this processor type, but it relies on an IP core library to fill in the remaining functional part of the node.

For the MicroBlaze and PowerPC processor types, the sizes of the data and program memories should also be specified. For each processor, external communication ports can be specified. In line 9 of Figure 2.8, an On-chip Peripheral Bus (OPB) port is specified for processor `MB_2`.

Additional peripherals can be defined using the `peripheral` element. On lines 12–14 of Figure 2.8, a Universal Asynchronous Receiver/Transmitter (UART) is instantiated. This component can be used as a low-bandwidth communication link between a processor on the FPGA and an external host, for example. Like the other components, the UART should be given a unique name; in this case it is called `UART_1`. On line 13, it is connected to the OPB bus using an OPB port.

In order to connect different components to each other, *links* can be used. A link connects exactly two components. On lines 16–19 of Figure 2.8, a link is used to connect MicroBlaze

processor MB2 to the UART peripheral UART_1. This way, the MicroBlaze and the UART can exchange data via the On-chip Peripheral Bus. For the FIFO channels that are present in the application specification, ESPAM automatically instantiates appropriately configured Fast Simplex Link (FSL) components. This is a data exchange interface with FIFO semantics available on Xilinx platforms. Any processor type can read data from and write data to an FSL component.

Mapping Specification

The mapping specification maps the processes of the application specification onto the processors of the platform specification. In some cases, the mapping specification can be left empty such that ESPAM automatically derives a mapping. This is allowed when, for example, no links are present and the platform consists of only one processor. If the mapping specification can not be left empty, the user has to provide it in the form of an XML file.

```
1  <mapping name="exampleMapping">
2
3     <processor name="MB_0">
4         <process name="ND_0" />
5     </processor>
6
7     <processor name="MB_1">
8         <process name="ND_1" />
9     </processor>
10
11    <processor name="MB_2">
12        <process name="ND_2" />
13    </processor>
14
15 </mapping>
```

Figure 2.9: An example mapping specification for the ESPAM tool.

In Figure 2.9, an example mapping specification in XML is shown. This mapping maps the different processes of our example application discussed in Section 2.2.3 onto the processors of our example platform discussed earlier in this section. Each processor element contains a list of process elements, indicating which processes are mapped onto the processor. It is possible to map multiple processes on the same processor. In the example of Figure 2.9, process ND_0 is mapped on processor MB_0, ND_1 on MB_1 and ND_2 on MB_2.

2.3.2 Tool invocation

Once the application, platform and mapping specification are available, invoking ESPAM is pretty straightforward. The following command launches ESPAM, which then generates an XPS project according to the input specifications.

```
espam --platform example.pla --adg example.kpn --mapping example.map \
      --xps --libxps $ESPAM_LIBXPS_directory
```

Using the first three pairs of command line arguments, the platform, application and mapping specification files are selected, respectively. The `--xps` switch turns on XPS project generation and the `--libxps` argument specifies the location of the XPS library, which is needed during XPS project generation.

2.3.3 Output

The result of running ESPAM using the command line described earlier is an XPS project. The top level project directory contains the following subdirectories and files:

- `code/`: This subdirectory contains the (C) source code files that belong to the various microprocessors in the design.
- `data/`: This subdirectory contains platform-specific data, such as User Constraint Files (UCF).
- `etc/`: This subdirectory contains implementation settings and scripts.
- `pcores/`: This subdirectory contains data for the various IP cores that are used in the design. For each IP core a separate subdirectory is created. Such a subdirectory typically contains the following items:
  - `hdl/`: The HDL files belonging to the IP core, usually written in VHDL or Verilog.
  - `data/core.mpd`: The Microprocessor Peripheral Definition (MPD) file. This file defines the characteristics of the IP core, such as external ports.
  - `data/core.pao`: The Peripheral Analyze Order (PAO) file. This files lists the HDL files belonging to the IP core and the order in which they need to be analyzed.

- `system.mhs`: The Microprocessor Hardware Specification (MHS) file. This file describes the different components of the system, such as the processors, other peripherals and the interconnections between the various components.
- `system.mss`: The Microprocessor Software Specification (MSS) file. This file describes software-related aspects of the system, such as the drivers that are needed for a certain component.
- `system.xmp`: The Xilinx Microprocessor Project (XMP) file, containing general information about the project.

Assuming that all IP cores needed by the design are present, the system can be synthesized into a bitstream using the XPS tool. Subsequently, the resulting bitstream can be used to configure an FPGA such that the original application can be executed. This completes the KPNGEN/ESPAM design flow.

2.4 PICO

The PICO [2] tool generates an RTL implementation from a specification written in a subset of ANSI C. It allows one to evaluate multiple alternative implementations and can provide the designer with a list of Pareto optimal implementations, in terms of area and performance. Between various stages of the synthesis process, the intermediate results can be verified using simulations. The final result is typically a Pipeline of Processing Arrays (PPA), which implements the functionality of the original input specification. This PPA is composed of a set of configurable architectural IP cores.

A PPA consists of a configurable amount of Processor Arrays (PAs) that are placed in a pipeline, interconnected using FIFO buffers. Each PA consists of one or more Processing Elements (PEs). A processing element consists of a variable number of different functional units, such as adders and multipliers. In Figure 2.10, the typical hierarchy of a PPA is shown. Each PA originates from a loop (nest) at the top level of the C specification.

PICO tries to exploit parallelism at various levels. Using Figure 2.11, we illustrate the various levels of parallelism for the application code fragment shown in the upper right corner. The three for-loops of this code fragment are mapped to separate PAs; we refer to them by means of the associated function call inside the loop. In Figure 2.11a, inter-task parallelism, or task overlap, is depicted. In this context, a *task* is one entire execution of the application code. A new task can

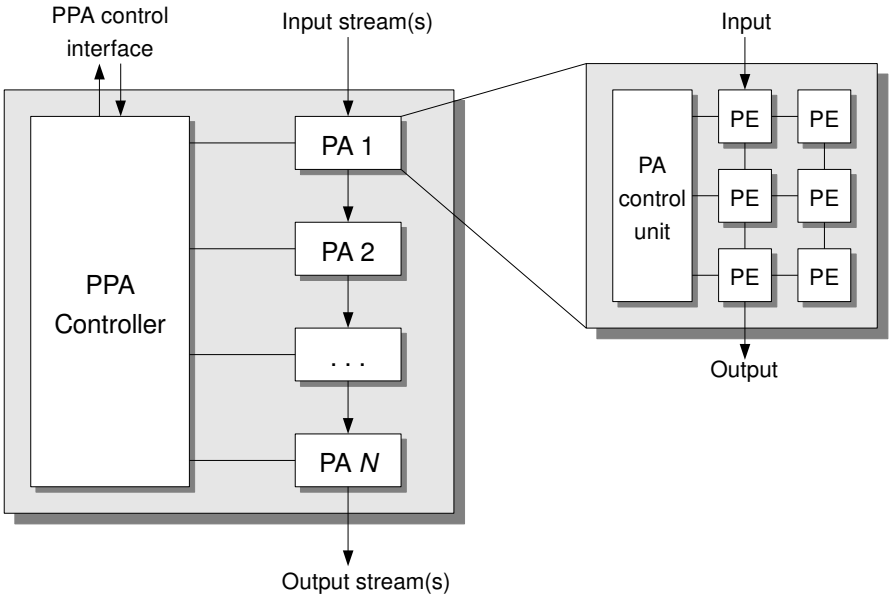


Figure 2.10: Overview of the PICO PPA hierarchy.

be started while the previous task is still running, typically when the first loop nest has finished. Task overlap is not obtained automatically: the user has to provide the appropriate input to the tool and/or the RTL result. In Figure 2.11b, intra-task level parallelism, or parallelism inside a single task, is depicted. Different portions of a task can typically be executed in a pipelined fashion. This is automatically exploited by the PICO tool. In Figure 2.11c, inter-iteration level parallelism is illustrated. Depending on the loop-carried dependencies of a loop (nest), iterations may also be executed in a pipelined fashion. Using software pipelining techniques, PICO obtains a schedule for the iterations. In Figure 2.11d, instruction level parallelism (ILP) is depicted. The low level operations of an iteration are scheduled in such a way that parallel execution is possible, while data dependencies of the original code are still respected.

2.4.1 Global Flow

First, we give a general overview of the PICO synthesis flow in this subsection. Subsequently, in the next subsections, we discuss the different key aspects in more detail.
The PICO Express application offers both a graphical and a command line user interface. Since

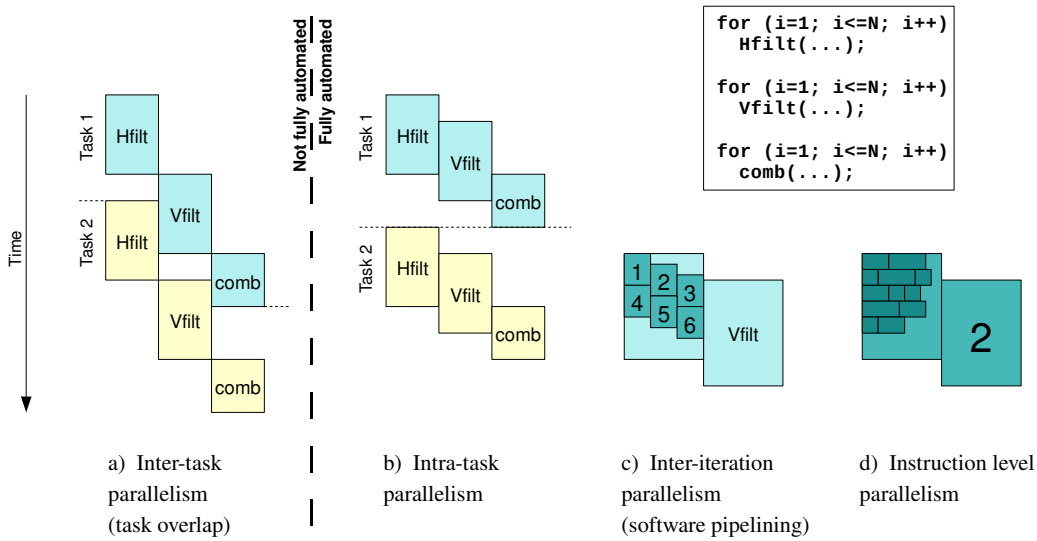


Figure 2.11: Different levels of parallelism that are taken into account by PICO.

the command line interface fits better in an automated design flow, we focus on the commands with which a certain operation can be performed.

Environment

For each design, a *project* has to be created. One possible way to achieve this is by creating a new directory, entering this directory and then launching the PICO Express tool from within this directory. PICO treats its initial working directory as project directory, so now one can configure the project. This is done using the `set_project_params` command, which accepts several arguments. Suppose we want to add a C source file called `my_func.c` to the project. The content of this C file is discussed in more detail in Section 2.4.2. The following command can be used to add the file to the project. Here we assume that this file already resides in the project directory.

```
set_project_params -sources my_func.c
```

The same command can be used to add C header files (`-headers` argument), input data files (`-data` argument) and result files (`-results` argument).

The next step is to create an *experiment*. An experiment represents one particular implementation of the project. Several experiments can be created, each with a different parameter

configuration. In order to create a new experiment, the following command is used:

```
create_experiment myexp
```

This command creates a new experiment called “myexp”. In the project directory, a new directory `myexp/` is created. This directory is used to store a copy of the input files, the experiment configuration and, after synthesis, the generated output. The `create_experiment` command also sets the new experiment as the current experiment. In order to switch to a different experiment, one uses the `select_experiment` command.

Configuring the current experiment is done using the `set_experiment_params` command. This command accepts a considerable amount of arguments. In this section, we only discuss the two arguments that are essential to the synthesis flow. In Section 2.4.3, we discuss some more arguments to this command. Suppose the top level function we want to synthesize into RTL resides in the file `my_func.c` and is called “func”. The following command can be used to configure the experiment accordingly:

```
set_experiment_params -appfile my_func.c -proc func
```

Synthesis

After configuring the experiment, the PICO synthesis process can be initiated. In order to convert the top level function to a PICO PPA core, several steps are required. These steps are depicted in Figure 2.12.

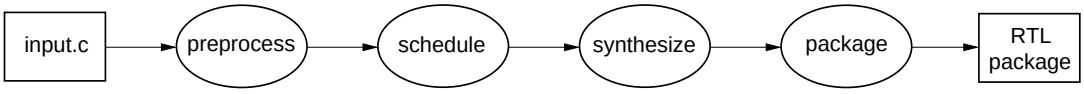


Figure 2.12: Global PICO Express flow.

First, the C file has to be preprocessed by PICO, using the `preprocess` command. During preprocessing, several source code transformations take place. For example, all function calls are inlined, such that the result of this phase consists of one function representing the entire functionality of the PPA. Also, statements that are not part of a loop nest are either moved to the nearest loop above or below, or placed inside a new loop with only one iteration. These restructurings

facilitate the construction of the PA pipeline in subsequent steps. In addition, syntactic checks are performed to make sure the input complies with the syntax of the PICO-C language.

The next step is the scheduling phase, which is invoked using the `schedule` command. During this phase, high level optimizations and loop optimizations are performed and loops are scheduled. The scheduling step is followed by the synthesis phase, invoked using the `synthesize` command. In this phase, instructions are scheduled, instruction level optimizations are performed and resources are allocated. Finally, the RTL implementation is written in the form of Verilog files. These files can be put into an RTL "package" using the `create_rtl_package` command. This command collects the Verilog files as well as reports, log files and simulation stubs, and places them in a conveniently arranged directory hierarchy on the file system.

Instead of issuing the `preprocess`, `schedule` and `synthesize` separately, the `build` command can be used to perform these operations sequentially.

2.4.2 C Input File

The (top level) C file that is provided to the `set_experiment_params` command using the `-appfile` argument is the starting point in a PICO design. The top level function in this file is synthesized to RTL. The structure of this file is similar to a regular C file, although there are some differences. In addition, PICO offers some constructs used for optimization and communication.

Communication Constructs

In order to allow data exchange between the PPA and its environment, several communication constructs are provided. A main distinction is made between *stream interfaces* and *live* scalar and array variables. Stream interfaces act as FIFO channels and can be internal, to communicate data between different loops, as well as external, to exchange data with the PPA environment. An internal stream is defined using the `FIFO` macro, which generates two access functions that can be called from within the PPA function:

```
1 FIFO(myFifo, int);
2
3 pico_stream_output_myFifo(y);
4 ...
5 x = pico_stream_input_myFifo();
```

At line 1, an internal stream "myFifo" is defined, which transfers data of the `int` type. At line 3, data is written into the stream; at line 5, data is read from the stream. The access functions for the

external streams are similar to those for internal streams. The declaration is somewhat different. Instead of using the `FIFO` macro, one has to declare the access functions manually. For an input stream, the `pico_stream_input_xxx` function has to be declared, where `xxx` should be replaced by the stream name. Similarly, for an output stream, the `pico_stream_output_xxx` function has to be declared. PICO automatically recognizes these functions as external stream interfaces and adds the appropriate external ports to the PPA RTL.

The global variables of a PPA file are treated as live variables. Variables that communicate input data to the PPA function are referred to as *liveins*; variables that communicate output data from the PPA function are referred to as *liveouts*. Scalar variables are bound to livein and liveout scalars, while arrays are bound to livein and liveout memories. The relation between the communication constructs and their RTL equivalent is discussed in Section 2.4.4.

Input Restrictions

Because the target architecture of the PICO tool set is very different from a regular von Neumann architecture, several restrictions are posed on the C input. Additional restrictions may arise from the fact that development is still going on, eventually leading to a relaxation of this class of restrictions.

For example, pointers are not allowed, as there is no notion of global memory. Recursive procedure calls and floating point operations are also not supported. The C input should be self-contained: all functions that are called have to be fully defined in the same or a directly included C file. In PICO Express version 08.01, that has been used during our research, composite data structures (using C's `struct` keyword) were not allowed, although the next release does support them. The loop nest structure of the top level function is also constrained: only *perfectly nested loops* are allowed. Note that this restriction applies to the C code obtained after preprocessing and function call inlining. Effectively, this restriction does not allow multiple loop statements inside another loop, as shown in the following example:

```
1     for () {        // Outer loop
2       for () {      // Inner loop 1
3         S1();
4       }
5       for () {      // Inner loop 2
6         S2();
7       }
8     }
```

Note that other non-loop statements are still allowed in the bodies of the loops. In this example, the second inner loop violates the loop structure restriction. In such a case, the user has to restructure the code to conform to the structural requirements again. If the iteration domains of both inner loops are equal and no data dependencies exist between statement `S1` and `S2`, the loops can be merged. Another option is to unroll one of the inner loops, although this quickly leads to increased hardware costs.

Source Code Annotations

To allow the user to provide PICO with additional information during synthesis, various source code annotation constructs are offered. Most of these annotations are specified via pragma directives. For example, to unroll a loop, the following pragma is placed directly before the loop that is to be unrolled:

```
#pragma unroll
```

Of course, the iteration count of the loop needs to be known at compile time if this pragma is applied. If the iteration count cannot be statically determined, the `num_iterations` pragma can be used to specify this number.

PICO already performs value analysis and bit width optimization to reduce the amount of bits needed for a variable. Using the `bitsize` pragma, the user can manually influence the amount of bits allocated to a certain variable, making more efficient variable sizing possible. Sizes of internal FIFOs can be controlled using the `fifo_length` pragma.

Typically, arrays are converted into local memories. Based on the size of an array, the local memory is either implemented using registers, or defined as an external SRAM. The user can override the default behaviour on a per-array basis. In order to synthesize an array as an internal register-based RAM, the `internal_fast` pragma can be used. The `user_supplied` pragma can be used if the user wants PICO to generate an external SRAM interface for a particular array.

Many other pragmas exist, which are described in the PICO Developer's Guide [49].

2.4.3 Implementation Settings

To influence the performance and behaviour of the resulting RTL, various settings can be configured and performance targets can be specified.

For RTL synthesis, a target clock frequency has to be chosen. This way, PICO will make sure the generated RTL meets timing constraints when processed by low level synthesis and implementation tools. To obtain such RTL, sophisticated constraint optimization algorithms are employed. If timing closure for a given clock speed can not be guaranteed, PICO RTL generation fails. Using the following command, one can set the target clock frequency to 100 MHz:

```
set_experiment_params -clock_freq 100
```

Using the Minimum Inter-Task Interval (MITI), one can influence the application performance. It represents the minimum number of cycles between two successive starts of a task, as depicted in Figure 2.13. PICO treats the specified MITI as a performance *goal*; the performance delivered by the hardware may be different. If a MITI value is specified, PICO computes loop Initiation Intervals (IIs) such that the rates of production and consumption are matched, resulting in small internal FIFO buffer sizes. This is known as *rate matching*. The user may specify some or all loop II values.

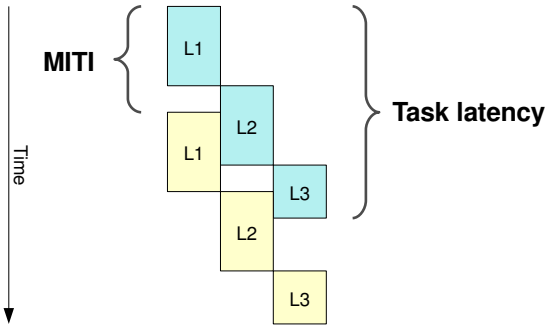


Figure 2.13: Execution of two tasks, illustrating the MITI concept.

If no liveins are present, the PPA can be configured to automatically restart itself after a task is completed. This has the advantage that no external PPA controller is needed to start each task. The auto-restart setting is turned on by passing an `auto_start_npa` argument to either the `synthesis` or `build` command. It is also possible to let the PPA run its first task "forever". This may be useful for real time streaming applications. This option is enabled by passing an `infinite_run_npa` argument to either the `synthesis` or `build` command.

2.4.4 RTL Output

As mentioned earlier, the `create_rtl_package` command creates a directory containing the RTL implementation of the PPA together with synthesis reports and simulation stubs. The following subdirectories which are relevant to our work can be found in the RTL package directory:

- `Logs/`: This subdirectory contains the log files of the preprocess, schedule and synthesis steps.
- `macrocells/`: This subdirectory contains the "macrocells" such as adders, multipliers and selectors that are instantiated by the PPA subcomponents. The macrocells are provided as Verilog files.
- `Reports/`: This subdirectory contains various reports, like the result of the rate matching steps, as well as detailed scheduling reports. The `report_summary.txt` report contains a summary of the most important PPA features, such as pipeline depth.
- `rtl/`: This subdirectory contains the modules of which the PPA is composed. The top level file is identified by the `_ppa` file name suffix. The PPA instantiates wrapped PA components (PAWs) which can be found in the files with a `_paw_N` suffix, where N is the corresponding PA number. The actual PAs reside in files identified by a `_pa_N` suffix. Likewise, each PE file is identified by a `_pe_N` suffix. All of the PPA components are provided as Verilog files.

The top level file contains the PPA module. This module has various input and output ports that are used for control and data communication purposes. A possible PPA module definition of an example auto-restart PPA, synthesized from a C specification with one input and one output stream, is shown in Figure 2.14. Lines 2–15 define the general control signals that are characteristic of a PPA. The `clk`, `reset` and `enable` signals are common system signals, expected to respectively provide a pulsating clock signal, a system reset signal and a signal enabling or disabling the component. Some of the remaining control signals provide an interface to the internal Processor Status Word (PSW), which is used to keep track of the PPA state. Such signals typically have a `psw_` prefix. For the first sixteen loops of a design, a stall signal is available which becomes high when the corresponding loop is forced to stall, due to unavailable inputs for example. In this example, we have only one loop of which the stall signal is shown on line 12. The `psw_busy` signal on line 13 indicates if the first stage of the PPA is busy or not. The `psw_init_done` and

psw_task_done ports become high when task initialization and task execution, respectively, have finished. Using the clear_init_done and clear_task_done ports, these status signals can be reset again. Because we solely use PPAs with the auto-restart feature turned on, the remaining signals on lines 5,6 and 9–11 are not relevant.

Lines 16–21 define the input and output stream interfaces. The names and amounts of these ports depend on the number of streams used in the application. Each stream interface typically consists of three ports. On lines 16–18, the interface for the input stream *myin* is defined. This interface consists of a data bus instream_myin_di_0, a signal instream_myin_req_0 which is raised when the PPA requests data, and a signal instream_myin_ready_0 which is high when data is available on the incoming data bus. On lines 19–21, the interface for the output stream *myout* is defined. This interface consists of a data bus outstream_myout_do_0, an outstream_myout_req_0 signal which is raised if the PPA requests to write data, and an outstream_myout_ready_0 signal indicating if the target of the output stream is ready to receive data. The protocol of the stream interfaces is rather straightforward. In order to read one element from an input stream, the req signal is held high for one clock cycle. If the ready signal is high at the rising edge of this clock cycle, the contents of the di bus are fetched. Otherwise, if

```
1   module myfunc_ppa(
2       clk,                        //  input  clk;
3       reset,                      //  input  reset;
4       enable,                     //  input  enable;
5       start_task_init,            //  input  start_task_init;
6       start_task_final,           //  input  start_task_final;
7       clear_init_done,            //  input  clear_init_done;
8       clear_task_done,            //  input  clear_task_done;
9       psw_livein_frames_in_use,   //  output [3 : 0] psw_livein_frames_in_use;
10      psw_liveout_frames_in_use,  //  output [3 : 0] psw_liveout_frames_in_use;
11      psw_released,               //  output psw_released;
12      psw_sa_0_stalling,          //  output psw_sa_0_stalling;
13      psw_busy,                   //  output psw_busy;
14      psw_init_done,              //  output psw_init_done;
15      psw_task_done,              //  output psw_task_done;
16      instream_myin_di_0,         //  input  [31 : 0] instream_myin_0;
17      instream_myin_req_0,        //  output instream_myin_req_0;
18      instream_myin_ready_0,      //  input  instream_myin_ready_0;
19      outstream_myout_do_0,       //  output [31 : 0] outstream_myout_do_0;
20      outstream_myout_req_0,      //  output outstream_myout_req_0;
21      outstream_myout_ready_0     //  input  outstream_myout_ready_0;
22  );
```

Figure 2.14: The Verilog module definition of a PPA.

the `ready` is low during this clock cycle, the `req` signal is held high and the PA stalls until the `ready` signal becomes high. The output stream interface works in a similar way.

2.4.5 Verification

At several points of the synthesis flow, PICO offers the possibility to verify the intermediate results, by means of simulations. This way, programming mistakes can be eliminated early in the design flow and the user can verify that the produced results behave as desired. In order to run the various simulations, a driver file written in C is needed. This file fetches input data, invokes the PPA procedure and reads back the resulting data. Input data is typically specified using *data* files and output data is specified using *result* files. For each simulation, PICO provides the data files to the driver code and compares the simulation result with the specified result files.

The first verification step is the *golden simulation*. This simulation is performed prior to any synthesis steps and is meant to verify the correctness of the input specification. In particular, the input/output relation is verified and a reference point is established. The next step after the preprocessing stage, is the linting simulation. This is a more thorough check, which checks the semantics of the C code and looks for undesirable runtime behaviour such as out of bound array accesses and uninitialized value usage. Also, a "bit accurate SystemC simulation" can be run at this point, for functional verification at the transactional level.

After the scheduling step, a "thread accurate SystemC simulation" can be performed. This simulation models the parallel behaviour of the hardware implementation allowing more accurate performance estimates. After RTL synthesis and packaging, RTL simulation can be performed in order to verify the behaviour of the resulting RTL. It is also possible to perform an RTL co-simulation such that one can study the interaction between the RTL and a host processor based on the driver code.

2.4.6 Tightly Coupled Accelerator Blocks

PICO also offers the possibility to synthesize a procedure into a *Tightly Coupled Accelerator Block* (TCAB). Such a TCAB can be integrated into a PPA or another larger TCAB. It allows for improved hardware sharing, leading to reduced hardware costs. If the entire TCAB procedure can be scheduled in one clock cycle, a purely combinational TCAB is synthesized. Otherwise, a pipelined TCAB is created.

Building a TCAB is done in the same way as a PPA is built, that is, one creates an experiment

```
1   module mytcab_paw_0(
2       clk,                          //  input  clk;
3       reset,                        //  input  reset;
4       enable,                       //  input  enable;
5       stallbar_in,                  //  input  stallbar_in;
6       stallbar_out,                 //  output stallbar_out;
7       op,                           //  input  op;
8       pred,                         //  input  pred;
9       instream_myin_di_0,           //  input  [31 : 0] instream_myin_di_0;
10      instream_myin_req_0,          //  output instream_myin_req_0;
11      instream_myin_ready_0,        //  input  instream_myin_ready_0;
12      outstream_myout_do_0,         //  output [31 : 0] outstream_myout_do_0;
13      outstream_myout_req_0,        //  output outstream_myout_req_0;
14      outstream_myout_ready_0       //  input  outstream_myout_ready_0;
15  );
```

Figure 2.15: The Verilog PAW module definition of a TCAB.

and synthesizes this experiment. The procedure that has to be synthesized into a TCAB is selected using the `-proc` argument of the `set_experiment_params`. An additional command is needed, to indicate that a TCAB should be built instead of a PPA:

```
set_experiment_params -build_tcab
```

In order to use this TCAB inside a PPA or another TCAB, no changes to the source code are needed: the TCAB procedure is invoked like any other C function. To build a component that makes use of a TCAB, a new experiment has to be created and the TCAB should be imported as follows:

```
set_experiment_params -import_tcab "mytcab"
```

Because multiple loop nests are not allowed in a TCAB procedure body, only a single processor array will be created. Hence, the RTL implementation of a TCAB consists of a single Processor Array Wrapper (PAW) component at the top level. A PAW module has a port definition similar to a PPA, although the control interface is different. A possible PAW module definition for a pipelined TCAB with one input stream and one output stream is shown in Figure 2.15. The generic system signals on lines 2–4 and the stream interface signals on lines 9–14 are similar to those of a regular PPA module. The main differences are found in the control signals on lines 5–8. Using the `stallbar_in` and `stallbar_out` ports, stalls of the PAW can be controlled. The `op` and `pred` signals need to be raised in order to start PAW operation.

```
Output/Release Latencies:
+----------------------+------------------+--------+------------------------>
| Architectural Latency | Physical Latency |  Type  | Item                    |
|  Adjusted[Specified]  | Adjusted[Initial]|        |                         |
+----------------------+------------------+--------+------------------------>
|          1[]          |        1[1]      | Stream | instream_in_1           |
|          3[]          |        3[3]      | Stream | instream_in_2           |
|          6[]          |        6[6]      | Stream | outstream_out_1         |
|         11[]          |       11[11]     | Stream | outstream_out_2         |
+----------------------+------------------+--------+------------------------>
```

Figure 2.16: An excerpt of the report showing the stream latencies.

A basic pipelined TCAB typically has a fixed pipeline depth, which is specified in the report file `report_summary.txt`. If the TCAB has more than one input stream, data from those streams is not necessarily fetched during the same cycle. The same applies to output streams. For example, consider a TCAB with a pipeline depth of 11, two input streams and two output streams. Data from the first input stream may be read immediately, while data from the second input stream is read during the third cycle. Likewise, data to the first output stream may already be written during the sixth cycle, while the second output stream is written during the last cycle of the pipeline. These latencies are specified in the `report_summary.txt` report file. In Figure 2.16, an excerpt of this file is shown, illustrating the scenario sketched above. In this figure, the Adjusted Physical Latency values correspond to the final latencies of the RTL. In Figure 2.17, we illustrate how this information is used when launching two successive tasks on the TCAB. For input stream `in_1`, data for the first task has to be available on the `in_1` data bus at cycle 1. Data for the second task has to be available on this bus at cycle 2. For the other input and output streams, the appropriate clock cycles are marked in a similar way.

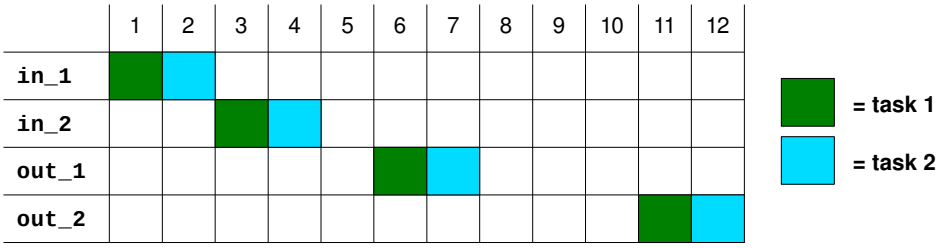


Figure 2.17: Timing diagram for successive execution of two tasks on the example TCAB discussed in this section.

Chapter 3

Hardware Node Models

In this chapter, we propose two hardware node models. Both models make use of PICO in a different way. Using these models, a KPN node can be synthesized into a completely functional hardware implementation. The first model is fairly straightforward, but its applicability is limited. The second model supports a broader input domain and can be applied to a larger set of applications.

3.1 Approach 1: PPA Hardware Node

The first model is based on a PICO PPA encapsulated in a small wrapper. The goal of this wrapper is to allow the PPA to be integrated into the existing (ESPAM) KPN hardware infrastructure. The PPA is responsible for loop nest control, data input and output operations and the actual computation of the node.

3.1.1 Model Description

In Figure 3.1, the structure of a PPA hardware node is shown. Several generic "control" signals enter the PPA hardware node. Based on these control signals, the controller drives the appropriate control ports of the PPA. Because the PPAs are generated with the auto-restart option, the controller size is kept to a minimum. Its only purpose is to correctly drive the reset and enable signals. The different nodes of a KPN hardware implementation are interconnected via the FSL bus. These connections are made via the input and output ports of the node. The FSL communication inter-

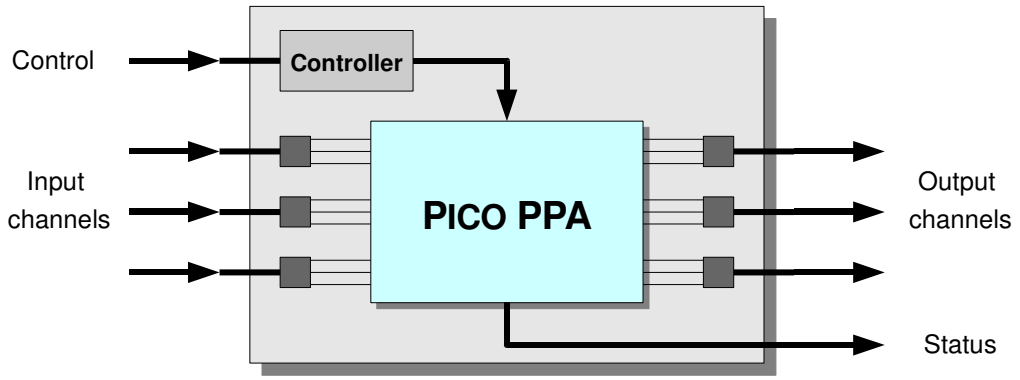


Figure 3.1: The PICO Hardware Node model, consisting of a PPA with additional wrapper logic.

face is very close to the PICO stream interface, although there are some minor differences. Using some additional logic, the FSL signals are appropriately connected to the PICO input and output stream ports. This is depicted by the dark squares that are attached to each channel. Finally, a status signal is provided, indicating if the computation of the node is completed. This status signal is driven by the `psw_task_done` signal of the PPA.

PPA Synthesis

The PPA of the hardware node is synthesized from an automatically generated PICO-C file. This C file consists of several parts, as depicted in Figure 3.2. Self-evidently, this file conforms to the PICO-C syntax and contains the top level function that is to be synthesized. In Figure 3.2, the different parts of the top level function are labelled with numbers *1* to *5*. First, loop iterators and data variables are declared (*1*). Next, the for-statements that iterate over the iteration space of the node are provided (*2*). In order to prevent synthesis problems, the bounds of these loops should be constant expressions that can be evaluated at compile time. Because of the SANLP nature of a KPN node process, only a single loop nest of a certain depth has to be synthesized using PICO. This fully conforms to the input that can be handled by PICO.

The loop nest body consists of three different stages, namely a read stage which reads data from the input channels, an execute stage which performs the actual computational workload of the node, and a write stage which writes the calculated data to the output channels. First, the input operations are specified (*3*). For each input port of the node, a `pico_stream_input`

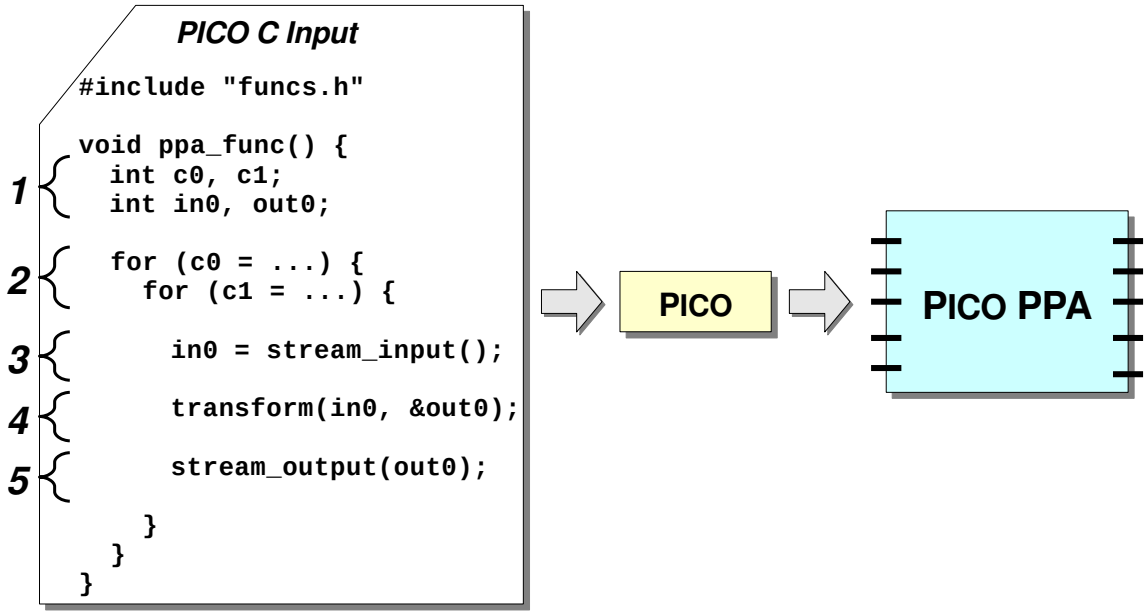


Figure 3.2: Generating a PICO PPA for integration in the PPA Hardware Node model.

call is generated. The results of these calls are stored in the input data variables. The input port operations are optionally guarded using if-statements to make sure they are issued during the appropriate iterations. The exact conditions of these guard statements are extracted from the KPN specification. Next, the node function call is performed (*4*). The implementation of this function has to be directly "visible" to the caller and will be inlined by the PICO preprocessor. All input and output data is transferred via the function arguments. The input arguments are listed first, which are the data variables with an `in` prefix. Each iteration, each of those variables is assigned a value by one of the previous input port operations. For the function output, the addresses of the output data variables (which are prefixed with `out`) are passed as arguments. At first, the use of the address-of operator (&) might seem inconsistent with the PICO-C syntax, but during function call inlining the address-of operator is eliminated and the inlined function body can directly write its output to the output data variables. Finally, the output operations are performed (*5*). This step is similar to the input operations step. For each output port of the node, a `pico_stream_output` call is generated, which outputs an output data variable. Again, these statements can be guarded using if-statements to make sure data is written during the appropriate iterations.

3.1.2 Restrictions

The PPA hardware node model is strongly dependent on the PICO tool, because both control flow and functionality of the node are generated using PICO. This implies that the applicability of this model is restricted by the set of input specifications that can be handled by the PICO tool. For example, the perfectly nested loop requirement that is mentioned in Section 2.4.2 makes it almost impossible to use loop statements in the node function implementations. This is due to the fact that the function call itself is already inside a loop nest, namely the loop nest that iterates over the iteration space of the node.

Another drawback is that a perfectly valid PICO-C program may still turn out to be unsynthesizable. This may happen when PICO can not guarantee timing closure for the implementation at a specified clock frequency. KPN nodes with more complex functions that result in an RTL implementation requiring multiple clock cycles per iteration often lead to synthesis failure.

The model is also not suitable for KPNs that include feedback loops or nodes with self-loops. This is caused by the pipeline behaviour of a PICO PPA. To illustrate this problem, consider a node *P1* that sends a token to node *P2* every iteration and then reads a token back from *P2*. Assume that node *P2* reads a token, performs an operation on this token and then outputs the transformed token. Effectively, there is an anti-dependence between the write and read operation of node *P1*. Unfortunately, expressing this anti-dependence in PICO-C is not trivial and our attempts to do so resulted in synthesis failure. If the anti-dependence is not made explicit in the PICO input, a deadlock occurs in the resulting network. This happens because, when the pipelines of the nodes are deep enough, the read operation of node *P1* stalls while the previous write operation is still pending in the pipeline. The stall of a single read operation typically leads to a stall of the entire PA, so none of the pending operations of node *P1* will make progress until the stalled read operation succeeds. Since the (pending) write operation of *P1* is not yet completed, node *P2* will not receive data and thus cannot produce a token that allows *P1* to continue. At this point, none of the nodes can make progress anymore.

3.2 Approach 2: TCAB Hardware Node

For the second model, we build further upon the LAURA processor model that was proposed in [50]. This processor model consists of a part that handles data communication and other control tasks, and an IP core insertion point where the actual computations of the node take place. Using

PICO, we generate a TCAB which implements the desired functionality of the node. This TCAB is then placed in the IP core insertion point. This results in a hardware node that is fully functional. Moreover, the TCAB hardware node model is more robust than the PPA hardware node model, as the different pipeline behaviour does not lead to undesired node stalls in the case of input unavailability.

3.2.1 Model Description

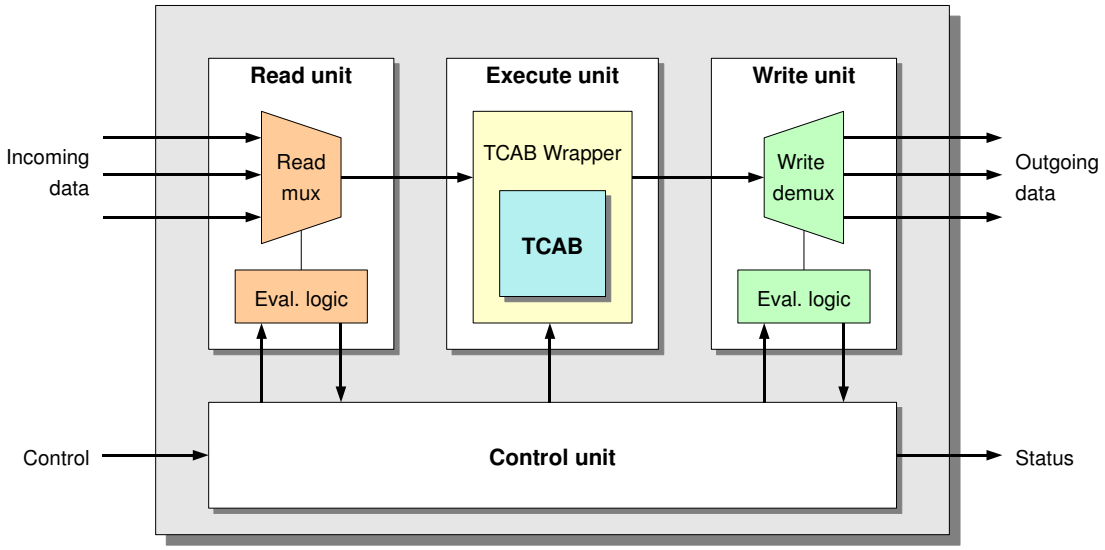


Figure 3.3: The TCAB Hardware Node model, consisting of a TCAB integrated in a LAURA processor.

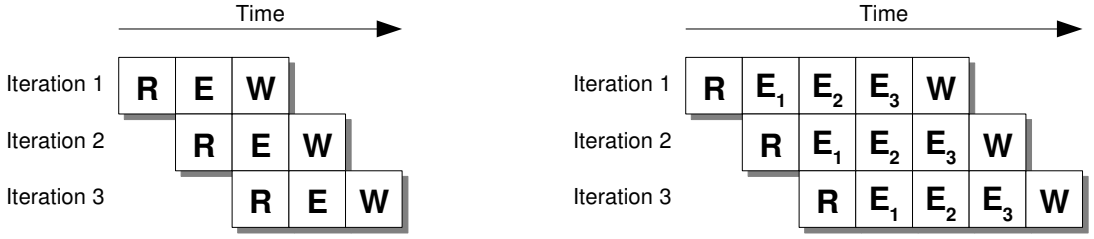
In Figure 3.3, we show a schematic overview of a TCAB hardware node. The node can be broken down into four different components, namely a read unit, an execute unit, a write unit and a control unit. The read unit is responsible for accepting the incoming tokens and passing the appropriate data at the right clock cycle to the execute unit. To do so, it keeps track of the current iteration by means of counters. Depending on the current iteration vector, data from the appropriate input ports is selected and forwarded to the execute unit. The execute unit implements the functionality of the KPN node function. Using its input data port, the input arguments are transferred to the hardware implementation of the function. Similarly, the output data port corresponds to the output arguments of the node function. The data produced by the execute unit is sent to the write unit.

The write unit is implemented in a way similar to the read unit. Depending on the current iteration vector, the data is written to the appropriate output ports of the hardware node.

Pipeline Model

Our TCAB hardware node model makes use of a pipelined execution model. In Figure 3.4, this execution model is visualized. Figure 3.4a shows a three-stage pipeline, which consists of a read stage (R), an execute stage (E) and a write stage (W). Figure 3.4b shows a five-stage pipeline, which consists of a read and write stage and three execute stages E_1, E_2 and E_3. This execution scheme applies to a situation where the TCAB is implemented using a three-stage pipeline.

The previous pipeline figures assume that for each read operation, the required data is already available. This leads to the optimal scheme, where a new iteration is initiated every clock cycle. However, this is not always the case, because the presence of self-loops or dependence on other nodes might lead to a situation where data is not yet available at the start of a new iteration. In such a case, the read operation should block until the data becomes available, while the operations already in the pipeline should be allowed to complete. This is achieved by inserting "bubbles" in the pipeline as illustrated in Figure 3.5. First, iteration 1 is started successfully. Now assume iteration 2 depends on the output data of iteration 1. This output data becomes available after the write stage of iteration 1. Hence, iteration 2 cannot be started immediately at the next cycle, and is delayed until after the write stage of iteration 1. Meanwhile, dummy data is sent to the execute unit, in order to keep the pipeline of the TCAB filled, thereby allowing the pending iterations to



a) The optimal execution scheme of a hardware node with a three-stage pipeline.

b) The optimal execution scheme of a hardware node with a five-stage pipeline.

Figure 3.4: The TCAB Hardware Node execution pipeline.

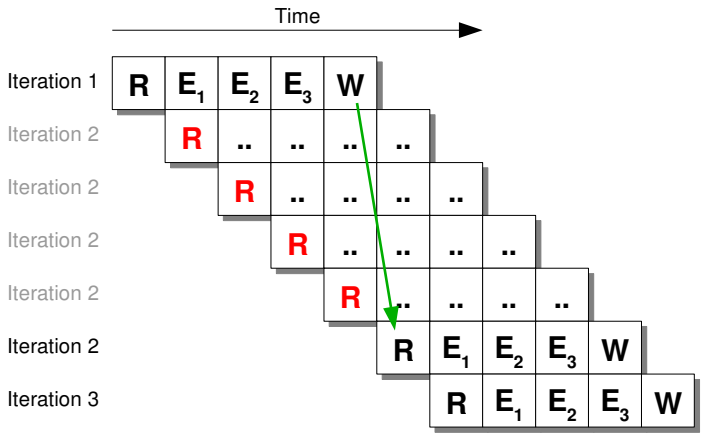


Figure 3.5: A TCAB hardware node pipeline scheme with bubbles inserted to account for unavailable data.

complete. The output data that is produced from the dummy data is discarded as soon as it leaves the execution unit. That is, once such a dummy token has made its way through all of the execution unit pipeline stages, it is discarded and no channel write operations take place.

Control Unit

The control unit is a key component for correct operation of the pipeline. A schematic overview of this unit and its connectivity with the other units is shown in Figure 3.6.
The read unit provides two signals to the control unit, namely the `EXIST` signal, which is high if all data needed for the current iteration is available, and the `DONE` signal, which is raised once the read unit has completed all iterations. Based on these signals, the control unit raises the `READ` signal in order to start a new iteration. This `READ` signal is passed to the read unit, thereby enabling the read unit to actually start reading the desired tokens. The `READ` signal is also transferred to a register, such that on the next rising edge of the clock signal, the execute unit is enabled via the `ENABLE_EX` signal. If the `ENABLE_EX` signal is high, the execute unit takes the data at its input port and passes it to the (pipelined) TCAB. For a TCAB with a pipeline depth of N, it takes N cycles before the result of the computation appears at the output port of the execute unit. This means that after N cycles, the data at the output port of the execute unit has to be passed to the write unit, which should subsequently write the data to the appropriate output channels. In order

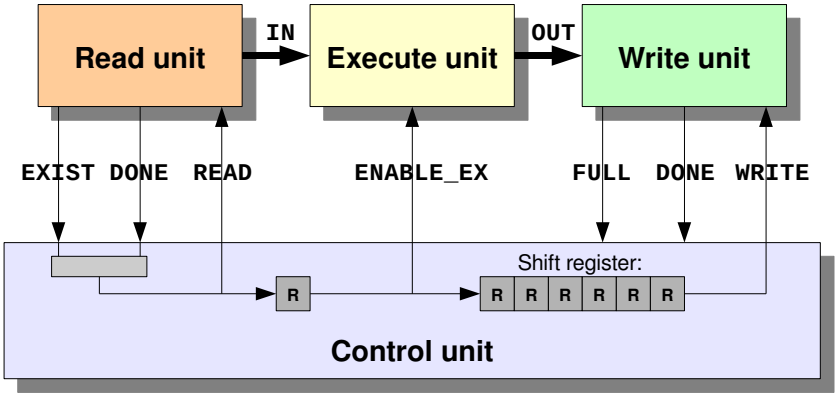


Figure 3.6: Schematic overview of the control unit of a TCAB hardware node.

to keep track of the iterations pending in the execute unit pipeline, a shift register of N bits is used. When a new operation is started on the execute unit, a one is pushed into the first position of the shift register. Otherwise, when no new operation is launched on the execute unit during the current cycle, a zero is inserted. At every clock cycle, the shift register contents are shifted one position. Consequently, at the other end of the shift register, a value is shifted out every clock cycle. Depending on this value, the `WRITE` signal is either kept low, in case of a zero, or raised, in case of a one. When the `WRITE` signal is high, the write unit writes the data coming from the execute unit to the appropriate output channels. After each write operation, the write unit updates its internal iteration counter(s). When all iterations have been completed, the write unit raises its `DONE` signal, which effectively means that execution of the entire node has finished. The write unit also provides a `FULL` signal, which becomes high if one or more output channels are full. The node does not accept new tokens if this signal is high, effectively realizing a blocking write condition.

Execute Unit

The execute unit contains the TCAB that implements the functionality of the node function. The pipeline depth N of this TCAB should be known after TCAB synthesis. The execute unit "consumes" the data at its input port(s) at clock cycle t and produces the result of the computation with this data at its output port(s) at clock cycle $t + N$. However, as we mentioned in Section 2.4.6, a TCAB does not necessarily take its input data at clock cycle t and produce the corresponding

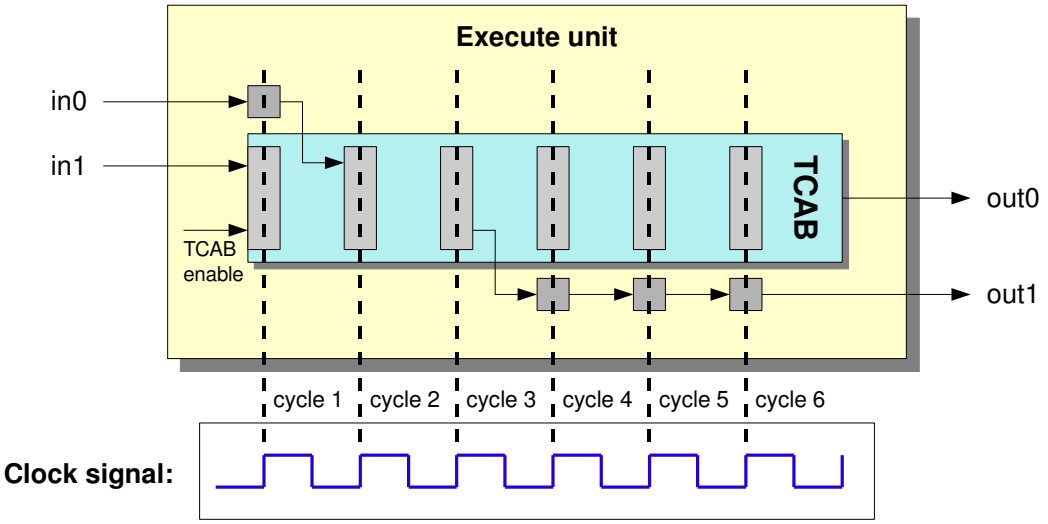


Figure 3.7: Schematic overview of an example execute unit with an integrated TCAB.

output at clock cycle $t + N$. Hence, the execute unit should account for these different input and output latencies in order to keep the execute unit consistent with the hardware node model. This is done by adding shift registers, as illustrated in Figure 3.7. Assume the TCAB fetches argument *in0* at cycle 2 and argument *in1* at cycle 1. Argument *in1* is fetched immediately at the start of the TCAB, so no additional actions are required. Argument *in0* however, is fetched one cycle after the TCAB is started, so we need to insert a transfer delay of one cycle. This is achieved by inserting a register, as indicated by the gray square in the upper left of Figure 3.7. For the output arguments, the problem is solved in a similar fashion. Assume argument *out0* is written at cycle 6, when the TCAB finishes its task, and argument *out1* is already written at cycle 3. For argument *out0*, no additional logic is needed. Transferring argument *out1* to the output port of the execute unit needs to be delayed three cycles to make it arrive exactly at cycle 6. This is achieved by inserting three registers after each other, as indicated by the three gray squares in the figure.

TCAB Generation

The PPA of the hardware node is synthesized from an automatically generated PICO-C file. This C file consists of several parts, as depicted in Figure 3.8 and contains the top level function that is to be synthesized as a TCAB. In Figure 3.8, the different parts of the top level function

are labelled with numbers *1* to *4*. First, data variables are declared (*1*). Next, the input arguments to the function are obtained from PICO input streams (*2*). For each input argument, a `pico_stream_input` call is generated. The node function call (*3*) is specified similarly as with the PPA hardware node model, as discussed in Section 3.1.1. Finally, the output produced by the function is written the the PICO output streams using `pico_stream_output` calls (*4*). Unlike the PPA of a PPA hardware node, a TCAB that is integrated in a TCAB hardware node has no notion of the node iterations. This results in a less complicated PICO-C top level function, which has no global loop nest or if-statements guarding the input and output operations.

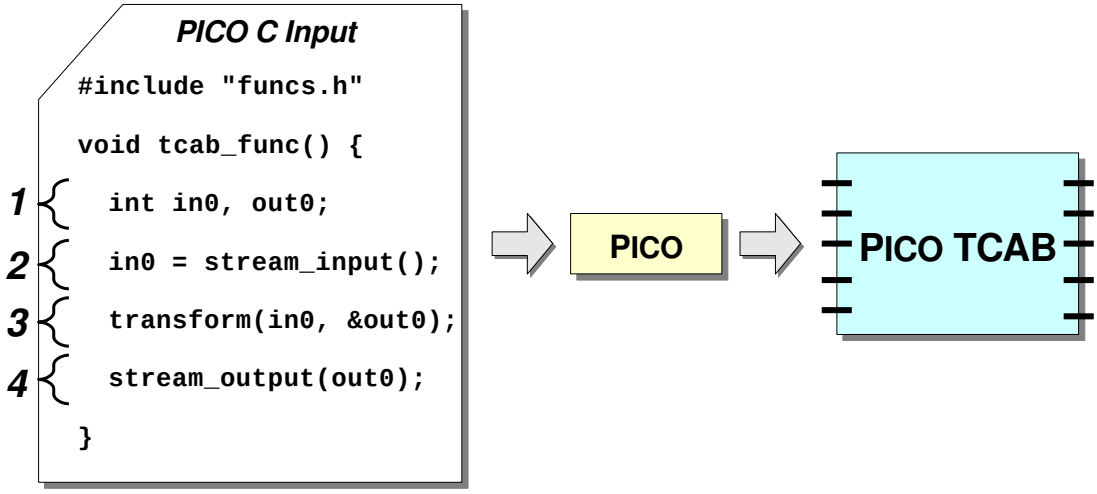


Figure 3.8: Generating a PICO TCAB for integration in the TCAB Hardware Node model.

3.2.2 Restrictions

Like the PPA hardware node model, the TCAB hardware node model depends on the PICO tool. This means the C specification of the node function is still subject to the restrictions of the PICO-C language, like lack of floating point arithmetic support. The TCAB hardware model presented in this section currently only supports TCABs with a constant pipeline depth that must be known before the HDL descriptions of the execute and control units are created. Furthermore, the TCAB must have an initiation rate of one, allowing a new task to be started every clock cycle. These limitations restrict the set of allowable C specifications for the node function. However, the TCAB hardware node model provides room for extension, at the expense of more complex control logic,

allowing TCABs with different characteristics to be integrated as well.

Chapter 4

Hardware Node Generation

In this chapter, we present the ESPAM-PICO tool. Using this tool, which is an extension to ESPAM, one can generate a hardware implementation of a KPN that contains one or more nodes of the types that have been discussed in the previous chapter. The key benefit of these new node types is that an external IP core library is no longer necessary, because the desired functionality can be generated directly from the C input specification.

4.1 The ESPAM-PICO Tool

Like the original ESPAM tool, ESPAM-PICO takes an application, platform and mapping specification and produces an XPS project. Additionally, a C file containing the node function implementations is needed, referred to as the *core functions* file.

4.1.1 Input

In Figure 4.1, the typical design flow of the KPNGEN and ESPAM-PICO tools is shown. In the upper part of the diagram, the four input files that have to be specified by the user are shown. The application is specified in a C file. This C file is processed by KPNGEN, resulting in an XML file containing the KPN specification of the application. The C file needs to conform to the syntax accepted by KPNGEN, that has been illustrated in Section 2.2.1. Of course it is also possible to bypass the KPNGEN tool, by providing a KPN specification directly to the ESPAM-PICO tool. Such a KPN specification can be written manually, or obtained from a different tool, like Compaan.

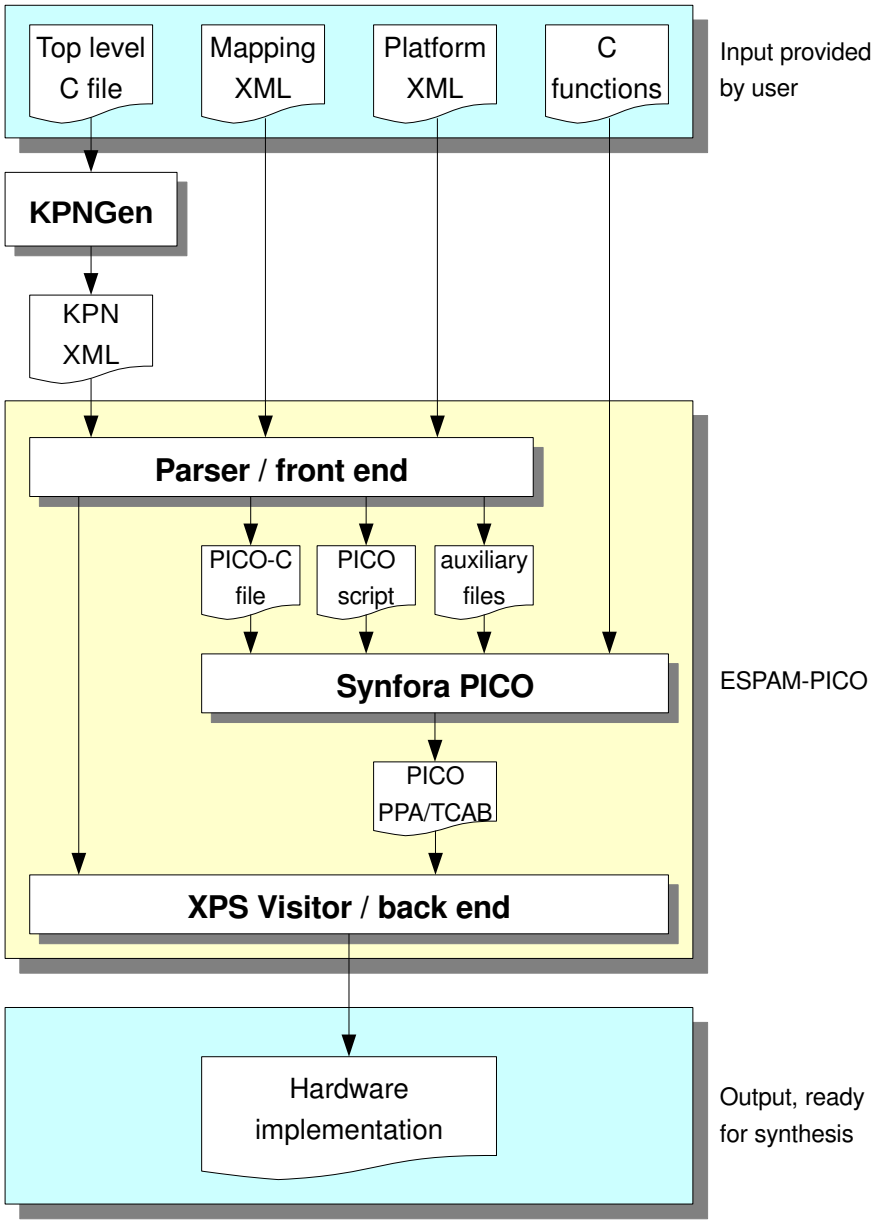


Figure 4.1: The KPNGEN/ESPAM-PICO design flow.

The mapping and platform specification formats for ESPAM-PICO are identical to those of ESPAM, which have been discussed in Section 2.3.1. ESPAM-PICO supports two additional processor types in the platform specification:

- **PicoPpaHWNode**: A PPA Hardware Node, according to the model discussed in Section 3.1.
- **PicoTcabHWNode**: A TCAB Hardware Node, according to the model discussed in Section 3.2.

For each processor of the *PicoPpaHWNode* or *PicoTcabHWNode* type, a C implementation of the function that is called by the node has to be provided. This is done by means of the core functions file, which is a separate C file that is directly given to ESPAM-PICO.

4.1.2 ESPAM-PICO Internals

In the middle part of Figure 4.1, the internal flow of the ESPAM-PICO tool is shown. The application, platform and mapping specifications are processed by the parser and internally modelled using abstract data structures. The traditional processor types like MicroBlaze and PowerPC are handled in the same way as ESPAM would handle them. The *PicoPpaHWNode* and *PicoTcabHWNode* processor types are handled differently. For each node of one of these types, a `pico/` directory is generated, which contains the following items:

- `aux_func.h`: A header file, containing macro definitions for e.g. minimum and maximum functions which might be used in loop bound or if-condition expressions.
- `core_funcs.c`: A copy of the core functions C file that was provided by the user.
- `genrtl.tcl`: A TCL script containing PICO commands. Using this script, a PICO project is created, configured and synthesized. The following is an example of a script with which a PPA can be built:

```
1     set_project_params -sources "HWN.c core_funcs.c"
2     set_project_params -headers "aux_func.h"
3     create_experiment imp000
4     set_experiment_params -appfile HWN.c -proc hwn_func
5     set_experiment_params -clock_freq 100
6     build -auto_start_npa
7     create_rtl_package
```

On lines 1 and 2, the source and header files are added to the PICO project. On line 3, an experiment is created, which is configured on lines 4 and 5. Next, the PPA is synthesized using line 6 and an RTL package is created using line 7. For a TCAB hardware node, the script is slightly different: an additional command is needed:

```
set_experiment_params -build_tcab
```

Furthermore, the `-auto_start_npa` argument is removed from the `build` command.

- `HWN.c`: The top level PICO-C file that is to be synthesized. The `aux_func.h` and `core_funcs.c` files are included at the beginning of this file. The remainder of this file follows the structure according to the desired hardware node model, as discussed in Chapter 3.
- `launcher.sh`: A small launcher script which initializes the environment and invokes the `genrtl.tcl` script.

After generating the PICO-C file and the script files, ESPAM-PICO invokes the `launcher.sh` script and waits until PICO has finished RTL synthesis of either the PPA or the TCAB. ESPAM-PICO then parses the port list of the generated PPA or TCAB module and connects the relevant control and data ports to the appropriate wires of the network. This way, the PPA or TCAB is integrated into the HDL specification of the node.

4.1.3 Output

The output of the ESPAM-PICO tool is a Xilinx Platform Studio (XPS) project, containing all files necessary for synthesis of a bitstream that can be downloaded onto an FPGA. The output is placed in a directory hierarchy which is structured similar to the ESPAM output directory hierarchy that is discussed in Section 2.3.3. Like the other node types, all *PicoPpaHWNode* and *PicoTcabHWNode* nodes get their own subdirectory in the `pcores/` subdirectory. The HDL files for each node are placed in the `pcores/<corename>/hdl/` subdirectory. In the next subsections, we discuss the contents of this subdirectory for the new node types.

PPA Hardware Node

The `hdl/` directory contains both a `verilog/` and a `vhdl/` subdirectory. The output generated by PICO, which consists of macrocells and the PPA RTL implementation, is placed in the

`verilog/` directory. The `vhdl/` directory contains a single VHDL file, in which the PPA wrapper that has been discussed in Section 3.1.1 is specified.

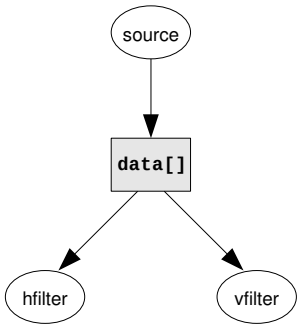
TCAB Hardware Node

The `hdl/` directory contains both a `verilog/` and a `vhdl/` subdirectory. The output generated by PICO, which consists of macrocells and the TCAB RTL implementation, is placed in the `verilog/` directory. The `vhdl/` directory contains the other components of the TCAB hardware node. These components, which have been depicted earlier in Figure 3.3, are placed in the following files:

- `controller.vhd`: Defines the control unit.
- `counter.vhd`: Defines a counter that is used by the read and write units to keep track of the current iteration.
- `eval_logic_rd.vhd` and `read_mux.vhd`: Together, these files define the read unit.
- `eval_logic_wr.vhd` and `write_demux.vhd`: Together, these files define the write unit.
- `execution_unit.vhd`: Defines the execution unit.
- `function.vhd`: Instantiates the TCAB and buffers input and output where necessary due to TCAB stream latencies. This component is instantiated by the execution unit.
- `hw_node_pack.vhd`: Contains common type and function definitions.
- `parameters.vhd`: Contains a mechanism that allows for run time adjustment of KPN parameters, as discussed in [51].
- `HWN.vhd`: Contains the top level component, which instantiates the read, execute, write and control units, and connects them accordingly.

4.2 Memory Model

In this section, we discuss two different memory models that can be used to communicate data between the nodes of a network.




```
int data[N];

for (i = 0; i < N; i++) {
  data[i] = source();
}

for (i = 0; i < N; i++) {
  hfilter(data[i]);
  vfilter(data[i]);
}
```

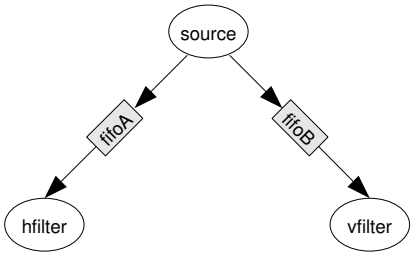
Figure 4.2: Illustration of a shared memory model. The `data` array is translated into a memory block of N words.

4.2.1 Conventional Memory Model

A straightforward hardware implementation of an arbitrary array in the input specification would be to instantiate a block of memory, as illustrated in Figure 4.2. In such an implementation, each element of the array is mapped to an address inside the memory block. The nodes that require access to the array are then connected to this memory block using address and data buses and control lines. Although this shared memory approach is easy to implement because of the close relation to the input specification, it is not an optimal method. Memories that are used for intermediate storage of computation results are often large and thus increase hardware cost. Particularly due to the array access patterns of stream processing applications, it is often unnecessary to keep the entire array available for arbitrary access all the time. Furthermore, a single memory block has a limited bandwidth, depending on the amount of read and write ports available. Increasing the amount of memory ports results in increased complexity and cost of the memory component.

4.2.2 Distributed Memory Model

Our KPNGEN/ESPAM-PICO tool flow makes use of a distributed memory model, which is a result of using the KPN model discussed in Section 2.1. Instead of instantiating a block of memory that is shared between multiple nodes, communication is implemented in a point-to-point fashion, as illustrated in Figure 4.3. This effectively means that storage of the array elements is distributed across the network. With the regular FIFO linearization model, a node writes array elements to an output channel in the same order as the successor node reads them. This way, only an



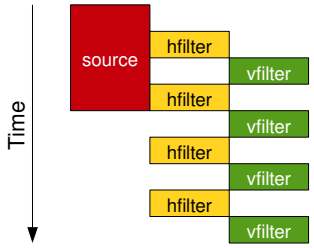

```
for (i = 0; i < N; i++) {
  x = source();
  fifoA.put(x);
  fifoB.put(x);
}

for (i = 0; i < N; i++) {
  hfilter(fifoA.get());
  vfilter(fifoB.get());
}
```

Figure 4.3: Illustration of a distributed memory model. The `data` array is replaced by two FIFO channels `fifoA` and `fifoB`.

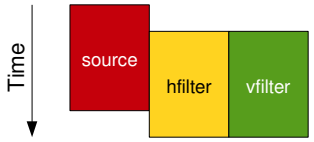
appropriately sized FIFO buffer is needed, which usually requires only a fraction of the amount of memory needed to store the entire array. When the same array element is read by multiple nodes, the element is effectively duplicated, as each consumer receives the data in one of its FIFO buffers. Although this "duplicated storage" increases overall memory usage, memory requirements are in most cases still considerably lower than with a shared memory implementation.

In some cases, when data is produced in an order different from the order in which it is consumed, the FIFO linearization model is not sufficient. For these cases, the approach described in [52] can be used, although this has not (yet) been implemented in ESPAM-PICO. In most cases, this approach still does not result in storage of the entire array in a large block of memory.

The difference between both memory models is illustrated in Figure 4.4. In Figure 4.4a,



a) Shared memory model: *hfilter* & *vfilter* nodes executing in an alternating fashion, due to accesses to a shared memory.



b) Distributed memory model: *hfilter* & *vfilter* nodes executing fully independent of each other.

Figure 4.4: Execution schemes for both memory models.

which corresponds to the scenario of Figure 4.2, the *hfilter* and *vfilter* nodes are executing in an alternating fashion. This is because the shared `data` memory from which they read can only handle one access at a time. When one node accesses the memory, the other node is stalled. In Figure 4.4b, which corresponds to the scenario of Figure 4.3, both nodes run fully in parallel, as they do not have to share access to the `data` memory anymore. Instead, each node receives the desired data directly from the *source* node. Due to the absence of stall cycles, shorter execution times are obtained.

High level languages like C typically assume a shared memory model. Deriving a distributed memory based implementation from such a high level input specification is not a trivial task. However, previous research has lead to systematic approaches, like Compaan [53] and KPNGEN [46], of which the latter is used in our tool chain.

In the XPS hardware implementation of the KPN, the nodes of the network are interconnected according to the application specification using Xilinx Fast Simplex Links (FSL) [54]. This component provides a flexible data communication channel with FIFO semantics between any two processors or nodes. Based on component configuration parameters such as FIFO depth and implementation method, the FSL component is efficiently synthesized using logic blocks only, or using a combination of logic and block RAMs (BRAMs).

Chapter 5

Experiments and Results

We have applied the approach described in the previous chapter to two different applications, namely Sobel edge detection and QR decomposition. In this chapter, we discuss those applications, describe the setup of the experiments, and show the obtained results.

5.1 Experiment Setup

For both of the applications that we describe in the next section, we have made several hardware implementations using our methodology. In order to measure performance and verify the behaviour of those implementations we have used the Active HDL 6.1 simulator from Aldec, as well as a physical platform.

5.1.1 Target Architecture

We have used the Xilinx XUP-V2P development board as our physical platform. This board contains a Virtex-II Pro 30 FPGA (XC2VP30), together with several peripherals. The Virtex-II Pro device has the following characteristics:

- Contains 2 integrated PowerPC 405 cores.
- Contains 136 BRAMs of 16 kbit each.
- Contains 13696 slices, available for logic synthesis.

Some key features of the XUP-V2P development board are:

- Provides a 100 MHz system clock.
- Contains three Serial ATA (SATA) ports.
- Supports up to 2 GB of Double Data Rate (DDR) SDRAM.
- Contains one 10/100 Ethernet port.
- Contains one RS-232 serial port.

The FPGA is programmed from an Intel Pentium D machine running a Fedora GNU/Linux operating system, referred to as the *host system*. Using the Xilinx iMPACT tool and a Xilinx Parallel Cable IV, bitstreams are downloaded onto the FPGA using the JTAG interface. For verification and performance measurement purposes, we use a UART peripheral, which is connected to the RS-232 serial port. Using a serial-to-USB cable, a low-bandwidth communication link is established between the FPGA and the host system. To actually integrate the FPGA in a larger system for streaming data processing, one can use one or more of the high speed interfaces available on the XUP-V2P board.

5.1.2 Experiments

We have applied our KPNGEN/ESPAM-PICO approach to various realizations of the Sobel and QR applications. In order to obtain a communication independent performance metric, we make a distinction between a functional verification experiment and a performance experiment of each realization. The first is the unmodified output produced by our KPNGEN/ESPAM-PICO tool chain. This is done to verify whether the functional behaviour of the hardware result is equal to the behaviour of the original software specification. That is: for a given input, both the original application and the generated hardware implementation should produce exactly the same output. Communicating input data to the hardware implementation and reading output data back requires a high-bandwidth communication interface. Otherwise, the hardware implementation will not run at its maximal speed. In order to obtain the execution time of the hardware implementation without any external communication overhead, we perform a second experiment where “dummy” input data is generated by the hardware implementation itself and the output data is discarded. This way, the hardware implementation can run at its maximal speed.

In order to measure execution time, the final sink node of an application is instrumented with a small fragment of VHDL code. This code keeps track of the amount of clock cycles passed since the last reset signal by incrementing a counter on each rising edge of the clock signal. The clock counter updates cease as soon as the final node indicates that it has completed its task, that is, it has received all data from its predecessor node(s). The obtained clock counter value is then communicated to the host system.

Using the Xilinx EDK tools, we have generated design reports for each experiment in order to collect device utilization data. In particular, we have measured the amount of slices and BRAMs that were needed to implement the various components of the automatically generated designs. These two numbers provide an indication of the amount of gates needed for a physical IC implementation. In the device utilization statistics shown later on, we do not take the source and sink nodes into account. We do, however, include the FIFO channels from the source nodes and to the sink nodes in our statistics, as the amount and sizes of those channels vary depending on the approach used.

5.2 Applications

In this section we describe two applications with which we demonstrate our approach. The first application (Sobel) results in a relatively straightforward process network with uncomplicated procedures inside the processes, while the second application (QR) leads to a process network which contains self-loops and more complex process procedures.

5.2.1 Sobel Edge Detection

Sobel edge detection is a common image processing operation. Its purpose is to detect “features” in an image, that are typically found at locations where the image intensity changes abruptly. In Figure 5.1, a monochrome image is shown on the left. On the right, the result of the Sobel edge detection operation is shown. One can see that objects that stand out against the background, such as the pillars of the bridge, result in a “bright” output, whereas the smoother areas, such as the sky, remain dark in the result.

The Sobel operation consists of a convolution of two 3x3 kernels with the original image to determine approximations of the horizontal and vertical gradient of the image intensity function. This is done by sliding the corresponding kernel over the image in the horizontal and vertical

Figure 5.1: Original image (left) and the result of the Sobel operation applied to it (right).

direction, respectively. Assuming I is the original image and $*$ denotes the convolution operator, the gradient approximations J_x (horizontal) and J_y (vertical) are computed as follows:

$$J_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * I \quad \text{and} \quad J_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * I$$

For each pixel, the resulting gradient approximations are combined into an approximated gradient magnitude v:

$$v = \frac{|J_x| + |J_y|}{4}$$

To visualize the result of the Sobel operation, the v values are usually plotted as a grayscale image, like the right half of Figure 5.1. In Figure 5.2, a C implementation of the steps discussed above is shown. This code is taken as the basis for our experiments with the Sobel application.

In our experiments with the Sobel application, we have used the monochrome image of Figure 5.1. The image has a width of 280 pixels and a height of 200 pixels, accounting for a total of 56000 pixels. As we do not compute the gradient at the borders of the image, the result consists of $278 \times 198 = 55044$ pixels. In Figure 5.3, we show the KPN that was generated from the sequential input specification. The nodes that implement the actual Sobel operation are labelled with *gradient_X*, *gradient_Y* and *absVal*. As can be seen from the network, the *gradient_X* and *gradient_Y* nodes can operate in parallel, since there are no dependencies between them.

```
1   for (j=1; j < M-1; j++) {
2     for (i=1; i < N-1; i++) {
3       gradient( image[j-1][i-1], image[j][i-1], image[j+1][i-1],
4                 image[j-1][i+1], image[j][i+1], image[j+1][i+1],  &Jx );
5       gradient( image[j-1][i-1], image[j-1][i], image[j-1][i+1],
6                 image[j+1][i-1], image[j+1][i], image[j+1][i+1],  &Jy );
7       absVal( Jx, Jy,  &av );
8       // send av
9     }
10  }
```

Figure 5.2: The source code of the Sobel application kernel.

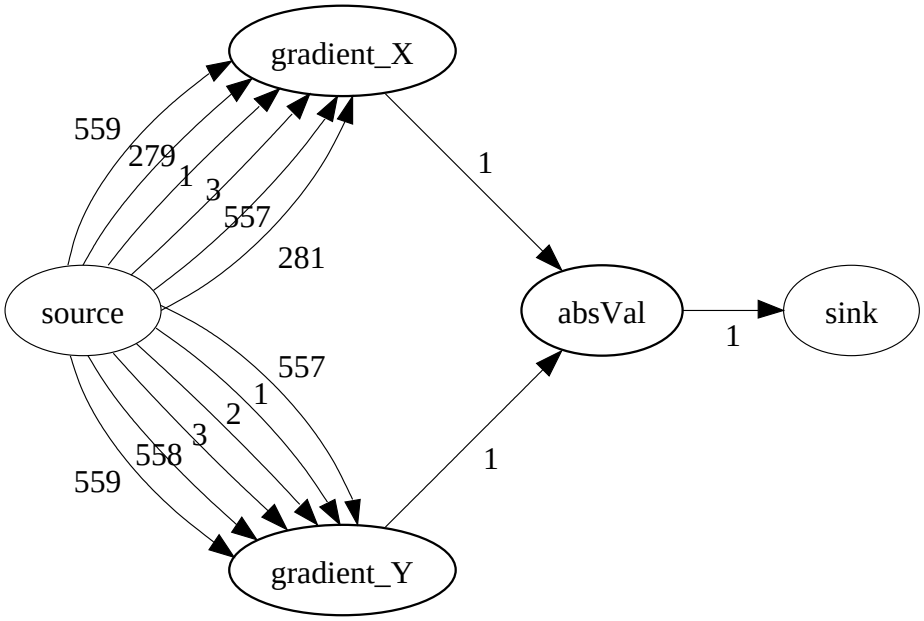


Figure 5.3: The automatically generated KPN for the Sobel application. The numbers next to the edges indicate the recommended minimum FIFO sizes.

5.2.2 QR Decomposition

The QR decomposition algorithm can be used to decompose a $K \times N$ matrix $\mathbf{X}$ into an orthogonal matrix $\mathbf{Q}$ and an upper triangular matrix $\mathbf{R}$. This operation can be used to find a least-squares solution for an over-specified set of linear equations, which finds applications in adaptive beamforming systems [55] for example. In our experiments, we use 21×7 matrices, that is, $K = 21$

and $N = 7$. A QR decomposition can be computed using various methods. The method we use is based on a series of Givens rotations [55].

The kernel code of our C implementation to compute **R** is shown in Figure 5.4. The *vectorize* function on line 3 computes an angle t and rotates a vector consisting of an element of **X** and an element of **R** through this angle. This way, the element of **X** is forced to zero. The *rotate* function on line 5 rotates a similar vector through an angle t computed earlier by the *vectorize* function. Using these operations, the **X** and **R** matrices are transformed until K Givens rotations have been performed. In the application mentioned above, the obtained **R** can then be used to obtain the least-squares weights.

```
1   for (k = 1; k <= K; k++) {
2     for (j = 1; j <= N; j++) {
3       vectorize( r[j][j], x[k][j],  &(r[j][j]), &(x[k][j]), &t );
4       for (i = j+1; i <= N; i++) {
5         rotate( r[j][i], x[k][i], t,  &(r[j][i]), &(x[k][i]) );
6       }
7     }
8   }
```

Figure 5.4: The source code of the QR application kernel.

```
1   vectorize(int r_in, int x_in,  int * r_out, int * x_out, int * t_out) {
2     int theta;
3     theta = -arctan2(x_in, r_in);
4     *r_out = r_in + cos(theta) * x_in - sin(theta) * r_in;
5     *t_out = theta;
6     *x_out = 0;
7   }
8
9   rotate(int r_in, int x_in, int t_in,  int * r_out, int * x_out) {
10    int cost = cos(t_in);
11    int sint = sin(t_in);
12    *x_out = cost * x_in  -  sint * r_in;
13    *r_out = sint * x_in  +  cost * r_in;
14  }
```

Figure 5.5: The integer-based source code of the *vectorize* and *rotate* functions.

The implementations of the *vectorize* and *rotate* functions are shown in Figure 5.5. Both functions invoke (a subset of) the trigonometric functions $\sin$, $\cos$ and $\arctan$. In order to implement these trigonometric functions, we have created two different implementations. The first implementation uses lookup tables to obtain an interpolated result of the functions. The second

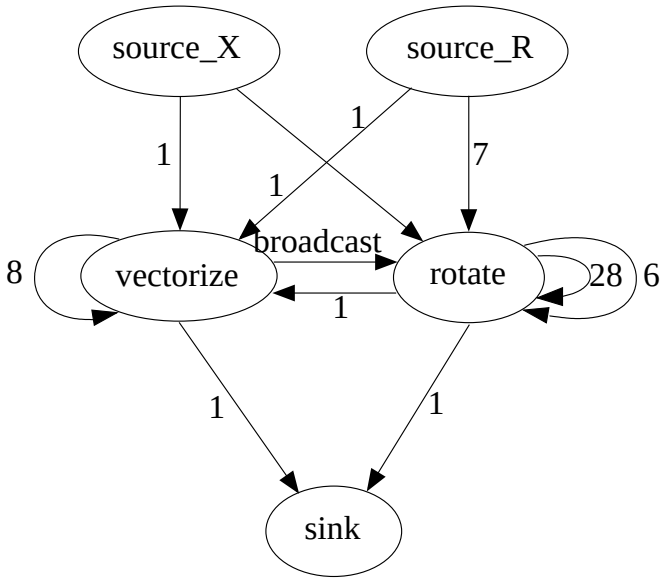


Figure 5.6: The automatically generated KPN for the QR application. The numbers next to the edges indicate the recommended minimum FIFO sizes.

implementation uses Taylor series to approximate the trigonometric functions. For these Taylor series based approximations, the following formulae have been used:

$$\begin{aligned} \sin(x) &= x - \frac{1}{6}x^3 + \frac{1}{120}x^5 \\ \cos(x) &= 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4 \\ \arctan(x) &= x - \frac{1}{3}x^3 + \frac{1}{5}x^5 \end{aligned}$$

That is, for each function, we compute the first three terms of the corresponding Taylor series expansion about 0. Such expansions are also known as Maclaurin series. Due to PICO lacking support for floating point arithmetic, we use integer operations for all computations. However, the structure of the C code remains equal if floating point operations are to be used.

In Figure 5.6, the KPN generated from the QR input specification is shown. The *source_X* and *source_R* nodes provide the **X** and **R** matrices, respectively. The *vectorize* and *rotate* nodes perform the actual computations and the *sink* node receives the final **R** matrix. The *vectorize* and

rotate nodes both contain self loops and both nodes are mutually dependent on each other. As mentioned in Section 3.1.2, these characteristics make the nodes unsuitable for implementation using the PPA hardware node model.

5.3 Results

In this section, we describe for each application the instances we have evaluated. Next, we discuss performance and device utilization statistics.

5.3.1 Sobel

We have generated hardware implementations and collected performance and device utilization data for the following instances of the Sobel application:

- Sobel-MBseq: The sequential code, implemented on a single MicroBlaze.
- Sobel-MBpar: Implemented using ESPAM as a network of MicroBlazes.
- Sobel-PICO: Implemented using PICO as sequential code. The input code is similar to the code we have sent through the KPNGEN/ESPAM flow. This means no PICO-specific annotations or constructs were used, hence one should not consider this as the most efficient PICO implementation.
- Sobel-HWN: Implemented using ESPAM-PICO as a network of PPA Hardware Nodes.
- Sobel-HWN-2: Implemented using ESPAM-PICO as a network of PPA Hardware Nodes. An unrolling transformation with a factor of 2 is applied to the *source*, *gradient*, *absVal* and *sink* nodes.
- Sobel-HWN-4: Implemented using ESPAM-PICO as a network of PPA Hardware Nodes. An unrolling transformation with a factor of 4 is applied to the *source*, *gradient*, *absVal* and *sink* nodes.
- Sobel-TCAB: Implemented using ESPAM-PICO as a network of TCAB Hardware Nodes.

We have also compared the experiments described above with two experiments that have been conducted during previous research, using different methodologies:

- Sobel-ESPAM: Implemented using ESPAM with hardware nodes that use custom IP cores. This experiment originates from the work of [50].

- Sobel-LAURA: Implemented using LAURA with hardware nodes that use custom IP cores. This experiment is described in [51].

Both designs were implemented on an ADM XRC-II board, which contains a Virtex-II FPGA (XC2V6000). The maximum clock frequency provided by this board is 66 MHz, which is different from the 100 MHz clock frequency offered by the XUP-V2P board.
The results for all implementations are shown in Table 5.1. The first column contains the name of the experiment. Next, the amount of FPGA slices and BRAMs needed for implementation are given. The fourth column contains the amount of clock cycles needed to process the 56000-pixel image. The fifth column contains the amount of images ("frames") that could be processed in one second by the particular implementation.

Setup	**Device utilization**		**Execution time**	**Throughput**
	Slices	2kB BRAMs	(cycles)	(frames/sec)
Sobel-MBseq	956	32	4717832	21
Sobel-MBpar	3397	55	2981813	33
Sobel-PICO	665	27	552385	181
Sobel-HWN	1226	7	56025	1784
Sobel-HWN-2	2768	14	28027	3567
Sobel-HWN-4	5860	28	14027	7129
Sobel-TCAB	1507	7	56030	1784
Sobel-ESPAM	1641	7	111440	897*
Sobel-LAURA	1710	7	223440	447*

Table 5.1: Synthesis and performance statistics for Sobel edge detection on a 280×200 grayscale image. An * in the throughput column indicates the value was scaled to 100 MHz.

Discussion

As we expected, the Sobel-MBseq experiment yields the lowest throughput. Execution times are large, because of the single thread of execution, the RISC nature of the MicroBlaze instruction set architecture and the general characteristics of microprocessors, which includes overhead of instruction fetching and decoding. Besides the processor's instruction pipeline, no other forms of parallelism are exploited. In the Sobel-MBseq experiment, the array containing the entire image is stored in the processor's local memory. Besides the limited bandwidth of 1 word per cycle of such a memory-based implementation, a considerable amount of BRAM components is needed to implement the local memory.

The Sobel-MBpar experiment yields a rather low throughput and a high device utilization. At the coarse-grained level, part of the computation can now be performed in parallel, which increases throughput by a factor of approximately 57 percent compared to the Sobel-MBseq implementation. However, each MicroBlaze processor still executes a single sequential thread of instructions, exploiting little or no fine-grained instruction level parallelism. Each node of the KPN results in the instantiation of a MicroBlaze core, which requires a considerable amount of slices. Moreover, for each MicroBlaze processor a memory is instantiated, which is composed of BRAMs. This leads to a high amount of BRAMs needed for the design. Although a MicroBlaze processor provides great flexibility in programming, the MicroBlaze implementation of the KPN is not the most efficient one.

By synthesizing the entire application using PICO, as we did in the Sobel-PICO experiment, the smallest implementation in terms of slice count is obtained. However, this implementation requires the user to provide an additional memory component for the array of the application code. For our implementation and input data, a memory of approximately 56000 bytes is required, equivalent to 27 BRAMs of 2 kilobytes. Moreover, the memory-based RTL result seems to limit throughput of the implementation significantly. This is probably caused by the use of 1-port memories, allowing only one read operation per cycle for a given memory. It should be noted that the code provided to PICO does not adhere to the PICO coding recommendations and does not make use of any PICO specific constructs such as internal streams. Hence, as a consequence, the experiment does not expose the full potential of PICO. A PICO hand design created by an expert user is likely to achieve a higher throughput and lower memory requirements.

Among the first four experiments, the Sobel-HWN experiment, which makes use of the PPA hardware node model, yields the highest throughput. Effectively, once buffers and pipelines are filled, the Sobel-HWN implementation is delivering one pixel per clock cycle. The implementation requires less than two times the amount of slices compared with the Sobel-PICO experiment. Due to the sizes of the FIFO channels of the KPN generated by the KPNGEN tool, memory requirements of the implementation are relatively high. Currently, we can not derive a KPN using KPNGEN which requires less than 7 BRAM components. A manually created line buffer based implementation created by an expert PICO user would require at most one BRAM for our input data, because at most approximately two lines of the image need to be stored. However, it should be noted that the Sobel-HWN implementation was automatically generated from plain C code containing no sophisticated constructs to expose parallelism.

Using the Sobel-HWN2 and Sobel-HWN4 experiments, we demonstrate the unrolling trans-

formation that was discussed in Section 2.1.1. In the device utilization statistics, we include source and sink nodes that are additionally needed for the unrolling transformation. As one can see in Table 5.1, device utilization roughly increases with a factor equal to the unrolling factor. The same holds for the throughput of an unrolled application instance. Thus, using the unrolling transformation one can increase throughput at the expense of increased hardware resource costs.

Using the Sobel-TCAB experiment, we compare both hardware node models that have been discussed in Chapter 3. Both models achieve the same throughput for the Sobel application. From the slice count statistics, we can see that for the Sobel application, the TCAB hardware node model is approximately 23 percent more expensive than the PPA hardware node model. Part of this can be attributed to the presence of infrastructure for runtime parameter adjustment, which is not present in a regular PPA. Also, the extensions added to support a more flexible pipeline behaviour lead to an increase in slice count.

By comparing the Sobel-HWN and Sobel-TCAB experiments with the Sobel-ESPAM and Sobel-LAURA reference points, we show the relation to other approaches. The network topology is equal for these four implementations, resulting in equal BRAM usage. The Sobel-HWN experiment requires significantly less slices, but this can (partially) be attributed to the absence of infrastructure for dynamic parameter modification. In the other three implementations, this infrastructure is included, leading to higher slice usage. The Sobel-ESPAM and Sobel-LAURA implementations require respectively about two and four times more clock cycles than the Sobel-HWN and Sobel-TCAB implementations. This is because of several differences in the implementation at the system level. The Sobel-ESPAM and Sobel-LAURA implementations include an off-chip memory which stores the image. As delivering a pixel every cycle is hard to realize in such implementations, longer execution times are the result. Also, FIFO channels were implemented using different components instead of FSL components, which might lead to differences in slice usage. Because of the lower clock frequency of the Sobel-ESPAM and Sobel-LAURA platform, throughput is lower for these implementations, respectively 592 and 295 frames per second. In Table 5.1, we have scaled those values to resemble an implementation at 100 MHz.

5.3.2 QR

We have generated hardware implementations and collected performance and device utilization data for the following instances of the QR application:

- QR-LUT: Implemented using our ESPAM-PICO approach as a network of PICO TCAB IP

Hardware Nodes. The trigonometric functions are implemented using lookup tables.

- QR-LUT skewed: Implemented using our ESPAM-PICO approach as a network of PICO TCAB IP Hardware Nodes. A skewing transformation has been applied to the algorithm. The trigonometric functions are implemented using lookup tables.
- QR-TA: Implemented using our ESPAM-PICO approach as a network of PICO TCAB IP Hardware Nodes. The trigonometric functions are implemented using Taylor series based approximations.
- QR-TA skewed: Implemented using our ESPAM-PICO approach as a network of PICO TCAB IP Hardware Nodes. A skewing transformation has been applied to the algorithm. The trigonometric functions are implemented using Taylor series based approximations.

Attempts to synthesize the original sequential code using PICO were unsuccessful due to timing closure problems. In such a case, the designer typically needs to manually rewrite the input code in order to relax timing constraints. Because the QR KPN contains selfloops and backedges, an implementation using PICO PPA hardware nodes is not possible, as mentioned in Section 3.1.2.

In Table 5.2, the results of our experiments with the QR application are shown. Again, the first three columns contain the experiment name, slice count and BRAM usage, respectively. The fourth column contains the amount of clock cycles needed to compute **R** for one 21×7 input matrix. The fifth column contains the number of such operations that can be performed in one second.

Setup	**Device utilization**		**Execution time**	**Throughput**
	Slices	2kB BRAMs	(cycles)	(tasks/sec)
QR-LUT	1417	0	2306	43365
QR-LUT skewed	1798	0	522	191570
QR-TA	2705	0	4205	23781
QR-TA skewed	3075	0	798	125313

Table 5.2: Synthesis and performance statistics for QR.

Implementation	*vectorize*	*rotate*
Lookup Tables (LUT)	11	5
Taylor based Approximations (TA)	20	11

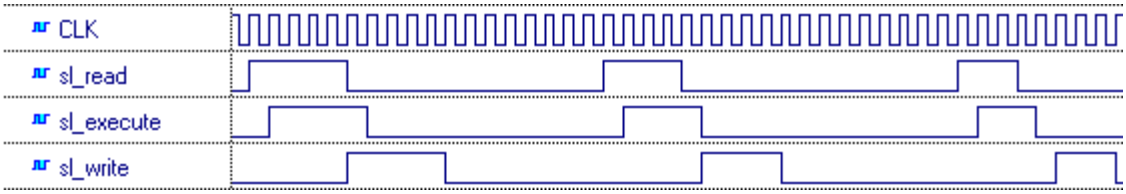
Table 5.3: Pipeline depths for the QR TCABs.

Discussion

Due to the small FIFO sizes, all FSL components are implemented using logic only. Hence, all implementations have zero BRAM usage. A first observation is that the lookup table based implementation is more efficient than the Taylor series based implementation, both in terms of device utilization and throughput. This is due to the higher complexity of the *vectorize* and *rotate* TCABs for the QR-TA implementation. As shown in Table 5.3, the increased function complexity results in a deeper pipeline for both TCABs. These higher pipeline depths of the QR-TA implementation can be considered as a closer match to a true floating point implementation.

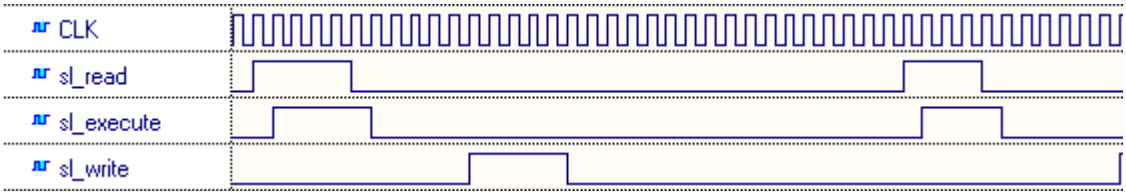
The skewed QR-LUT implementation requires about 27 percent more slices than the unskewed QR-LUT implementation. This can be attributed to more complex loop control and larger FIFO sizes. Throughput of the skewed implementation is increased by a factor of 4.4. This increase can be explained by looking at the signals that allow the read, execute and write unit of the *rotate* node to advance, which are `sl_read`, `sl_execute` and `sl_write`, respectively. In Figure 5.7a, a fragment of the simulation waveform containing these signals is shown for the QR-LUT experiment. First, the `sl_read` signal is raised in order to read the input data. In the next cycle, all data is read and the `sl_execute` signal is raised in order to start the execute unit. At the fifth cycle after the enabling of the execute unit, the `sl_write` signal is raised, as the execute unit containing the *rotate* TCAB has produced data that is ready to be written. Meanwhile, `sl_read` is kept high for another few cycles, as long as input data is still available. Unfortunately, at some point the *rotate* node sends data back to the *vectorize* node and requires new data from the *vectorize* node. The *vectorize* has to compute this new data using the data it just received from the *rotate* node. Because this occurs during two successive iterations, the *rotate* node will not receive the data immediately and is forced to wait until the data becomes available again. This point is visible in Figure 5.7a where the `sl_read` signal is dropped. Once the data is written to the *vectorize* node, this node performs its computation and sends new data to the *rotate* node. Meanwhile, the *rotate* node is completely idle, waiting for new data to arrive. Once it receives new data from the *vectorize* node, the `sl_read` signal is raised again, although only for a few iterations. Throughout execution of the application, this situation with stalls occurs, resulting in considerable underutilization of the pipeline.

In Figure 5.7b, the same signals are shown for the skewed QR-LUT implementation. In this waveform, the three signals remain high for almost all the time. In fact, besides some troughs during the first and last iterations, the signals remain high during execution of the entire application.

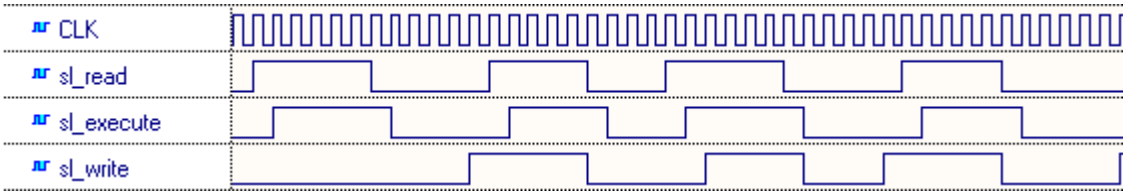


(a) QR-LUT

(b) QR-LUT skewed



(c) QR-TA



(d) QR-TA skewed

Figure 5.7: Fragments of the simulation waveforms of the read, execute and write unit enable signals for the *rotate* node of four different QR application instances.

This is caused by the high level skewing transformation on the complete KPN of the application, which leads to a different iteration execution order. Now, the *rotate* node executes other iterations while the *vectorize* node computes new data for the *rotate* node. This leads to a better utilization of the *rotate* node pipeline and a larger throughput.

For the QR-TA implementation, the skewed implementation requires about fourteen percent more slices than the unskewed implementation. Throughput of the skewed implementation is increased by a factor of 5.3. In Figure 5.7c, the `sl_read`, `sl_execute` and `sl_write` sig-

nals are shown for the unskewed QR-TA implementation. Due to the deeper TCAB pipelines, overall pipeline utilization of this QR-TA implementation has become worse than the QR-LUT implementation because the *rotate* node is now stalled for longer periods. In Figure 5.7d, simulation waveforms are shown for the skewed QR-TA implementation. The lengths of the troughs are much shorter than those in Figure 5.7c, indicating an improved pipeline utilization for the skewed QR-TA implementation. However, maximum pipeline utilization is not achieved, as the three signals are not kept high for long continuous periods of time, like we saw in Figure 5.7b. This is because the different iteration schedule can not "hide" the higher pipeline latency between the dependent iterations anymore. In order to further increase throughput, one could increase the problem dimensions or operate on multiple QR instances at the same time, as suggested in [56].

5.4 Design & Implementation Times

In this section, we show the amount of time needed to obtain a complete implementation of an application. We have measured this "design time" for both Sobel and QR. The results can be found in Table 5.4. All tools have been run on the same system, an Intel Pentium D at 3.4 GHz with 2 GB of RAM, and no workload other than the tools themselves was present during execution time measurement.

Step	**Design time**		**Manual / Automatic**
	Sobel-HWN	QR (LUT)	
1. Writing C file(s)	5 min.	10 min.	Manual
2. Writing .pla, .map	5 min.	5 min.	Manual
3. KPNGEN	5 sec.	5 sec.	Automatic
4. ESPAM-PICO	6:50 min.	5:05 min.	Automatic
5. Synthesis using XPS	12:45 min.	16:25 min.	Automatic
6. FPGA Configuration	90 sec.	90 sec.	Automatic
Total	31:10 min.	38:05 min.	

Table 5.4: Time needed to generate a bitstream starting from a C input specification.

The first step in the design flow is to write the top level C file of the application and the C file containing the implementations of the called procedures. For both applications, a sequential implementation of the top level C file was already available. For Sobel, the procedure implementations were already available. For QR, we have written a new lookup table based implementation

of the procedures. Because this step only involves C programming, only little time was needed.

For each application, platform and mapping specifications have to be created. In general, this is a trivial task, so for both applications no more than 5 minutes were needed to accomplish this task. For a straightforward one-to-one mapping of KPN nodes to PPA or TCAB hardware nodes, this step could be automated as well, although such functionality is currently not yet implemented. At this point, the required manual actions have been completed. The next steps are all automated and no user actions are required anymore.

As a third step, the KPNGEN tool is invoked. For both applications, the translation from a sequential to a parallel specification takes no more than five seconds. Next, the ESPAM-PICO tool is invoked. The running time of this tool strongly depends on the amount of times PICO needs to be invoked, which directly depends on the amount of PICO nodes in the platform specification.

Once the XPS project is complete, the Xilinx synthesis tools can be invoked to generate a bitstream. Of all steps, this fifth step typically consumes most of the time, since it includes the application of sophisticated place and route algorithms. Finally, the generated bitstream is downloaded onto the FPGA and results are read back. This takes about 90 seconds.

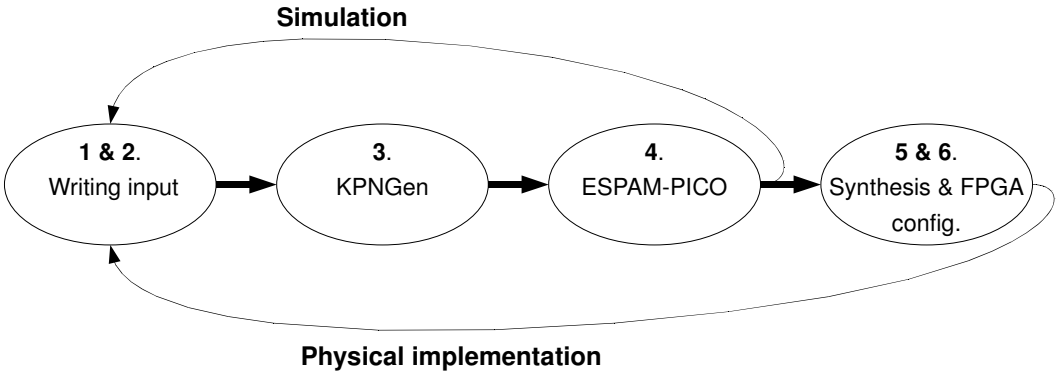


Figure 5.8: Feedback loops in the ESPAM-PICO design flow.

In a typical design flow, multiple iterations of the steps described above are needed before a final implementation is obtained. The output produced by the last step is used by the engineer to adjust the input specification in order to correct programming mistakes or achieve better performance. For productivity reasons, a short iteration time is desired; that is, the engineer should get feedback from the tools within a reasonable amount of time. Instead of synthesizing the design into an FPGA bitstream each time, it is also possible to perform simulations at the HDL level.

The HDL of the design is available after running ESPAM-PICO, as illustrated in Figure 5.8 by the “simulation” backedge. Simulation allows the engineer to skip the time-consuming bitstream synthesis step, leading to shorter feedback times. Particularly when no physical implementation is required, during application behaviour verification for example, simulation may already provide the information needed for input specification adjustment. Simulation is also especially useful to decide which high level transformations to apply.

Chapter 6

Future Work

The approach that has been discussed in the previous chapters is not a final and complete solution to the problem mentioned in the beginning of this thesis. Instead, our approach provides a foundation, that can be extended in order to support a wider range of applications and further increase performance and efficiency of the obtained implementations.

For example, consider the different channel types. Right now we are solely using PICO streams for data communication, but this is not always the most efficient solution. It might be interesting to use shift registers and live variables as well, which are also offered by PICO. Also, implementing self loops as regular FIFOs is not the most efficient method. By implementing self loops in a more sophisticated way, device utilization may improve significantly. The memory requirements are another point of concern, as KPNs with many and/or large FIFO channels result in large FIFO buffer memories. Reducing these memory requirements is important to obtain efficient implementations of larger applications or larger input data dimensions. This might be achieved by investigating how to increase self-reuse and reduce data duplication.

In terms of power efficiency, the TCAB hardware node model is not the most favourable solution. The integrated TCAB is always operating at full functionality, due to the insertion of dummy data when the node has to wait for new data. Unfortunately, we had to choose for this solution, as we have no control over the pipeline behaviour of PICO generated components. A better solution would be to stall the TCAB as well, after allowing the pending operations to complete.

At the front end, other improvements can be made. Currently, the designer has to separate the top level function and other functions in different files. A more robust front end eliminates

this issue. The relatively long running times of the ESPAM-PICO tool are caused by subsequent invocations of PICO. However, during design iterations, the input specification is often modified only slightly. In such a case, complete resynthesis of all hardware nodes is not always necessary. By employing a caching system, unnecessary PICO invocations can be avoided, leading to a shorter overall design cycle.

As new features are added to the PICO tool, the possibilities of the ESPAM-PICO tool also increase. For example, C struct support that was recently added to PICO could be added to ESPAM-PICO as well, in order to allow convenient handling of larger data blocks. Other relaxations of the PICO input restrictions might become directly available for use in the core functions file. In such a case, little or no adaptations to ESPAM-PICO are needed.

Selecting the appropriate transformations for a particular application is not trivial. Currently, the user has to specify these transformations manually. However, our flow is closed, meaning that once the appropriate input specifications are written it can run fully automated. This allows us to perform automated Design Space Exploration (DSE), where various instances of an application are generated and evaluated.

Chapter 7

Conclusions

In the previous chapters, we have presented a new approach for automated generation of RTL implementations from sequentially specified static affine nested loop programs written in the C language. This is achieved by combining the KPNGEN and ESPAM tools resulting from previous research with the commercial PICO tool of Synfora Inc. The resulting tool, which we call ESPAM-PICO, is capable of producing a complete RTL implementation of the application specification. This implementation is immediately ready for synthesis, in contrast to the regular ESPAM flow which requires the user to provide additional IP cores.

By using a distributed memory model, we can achieve significant speedups, compared to implementations that employ a shared memory model. Due to the characteristics of FPGAs, this distributed memory model fits particularly well to such platforms. Coarse-grained partitioning of the input leads to a set of smaller and less complex inputs for the subsequent fine-grained implementation stage. This allows our tool flow to accept a wider range of applications without the need of manual application code restructuring.

Also, by using the KPN model of computation, we can automatically apply transformations to the application, which allows us to generate various application instances with increased throughput at the cost of additional hardware resource usage. We have shown that by applying an unrolling transformation to the Sobel application, we can double throughput, at the cost of doubled hardware resource requirements. By applying a skewing transformation to two different implementations of the QR application, we can achieve throughput increases of factors of 4.4 and 5.3, at the expense of only 27 and 14 percent increases in hardware cost.

Currently, an important concern of our methodology are the memory requirements. Depending on the amounts and sizes of FIFO channels in a KPN, memory requirements can increase quickly. Further investigation of self-reuse and possible improvement of the KPNGEN tool may lead to reduced memory requirements for implementations generated with our ESPAM-PICO tool flow.

Bibliography

[1] D. Pham et al. The Design and Implementation of a First-Generation CELL Processor. In *ISSCC Digest of Technical Papers*, pages 184–5, 2005.

[2] Synfora Inc. PICO Technology. `http://www.synfora.com/`, last accessed: 2008-06-11.

[3] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers. Optimized Generation of Data-Path from C Codes. In *Design Automation and Test Europe (DATE'05)*, March 2005.

[4] Mentor Graphics. Catapult Synthesis. `http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/`, last accessed: 2008-08-08.

[5] C. Zissulescu, T.P. Stefanov, A.C.J. Kienhuis, and E.F. Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *Proc. of the 13th Int. Conference on Field Programmable Logic and Applications (FPL'03)*, pages 911–920, September 2003.

[6] Y.D. Yankova, G.K. Kuzmanov, K.L.M. Bertels, G.N. Gaydadjiev, J. Lu, and S. Vassiliadis. DWARV: DelftWorkbench Automated Reconfigurable VHDL Generator. In *Proc. of the 17th International Conference on Field Programmable Logic and Applications (FPL'07)*, pages 697–701, 2007.

[7] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

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

[9] E.A. de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proc. of the 15th International Symposium on System Synthesis (ISSS'02)*, pages 68–73. ACM Press, 2002.

[10] T.P. Stefanov, C. Zissulescu, A. Turjan, A.C.J. Kienhuis, and E.F. Deprettere. System Design using Kahn Process Networks: The Compaan/Laura Approach. In *Proc. of the Int. Conference Design, Automation and Test in Europe (DATE'04)*, pages 340–345, 2004.

[11] T.P. Stefanov, A.C.J. Kienhuis, and E.F. Deprettere. Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances. In *Proc. of the tenth international symposium on Hardware/software codesign (CODES'02)*, pages 7–12. ACM Press, 2002.

[12] H. Nikolov, M. Thompson, T.P. Stefanov, A.D. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E.F. Deprettere. Daedalus: Toward Composable Multimedia MP-SoC Design. In *Proc. of the ACM/IEEE Int. Design Automation Conference (DAC'08)*, 2008.

[13] Daedalus home. `http://daedalus.liacs.nl/`, last accessed: 2008-06-11.

[14] I. Page. Hardware-Software Co-synthesis Research at Oxford. In *Proc. of the IEE Vacation School on Hardware/Software Co-design*. IEE, July 1997.

[15] SpecC Technology Open Consortium. SpecC. `http://www.specc.org/`, last accessed: 2008-01-08.

[16] R. Dömer, A. Gerstlauer, and D. Gajski. *SpecC Language Reference Manual*, December 2002.

[17] D.K. Wilde. The ALPHA language. Technical Report 999, IRISA, January 1994.

[18] T. Risset et al. Alpha homepage. `http://www.irisa.fr/cosi/ALPHA/`, last accessed: 2008-01-17.

[19] D.K. Wilde and O. Sié. Regular Array Synthesis using Alpha. In *International Conference on Application-Specific Array Processors*, August 1994.

[20] S. Gupta, N.D. Dutt, R.K. Gupta, and A. Nicolau. SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations. In *International Conference on VLSI Design*, January 2003.

[21] S. Gupta et al. SPARK: High-Level Synthesis using Parallelizing Compiler Techniques. `http://mesl.ucsd.edu/spark/`, last accessed: 2008-01-10.

[22] Los Alamos National Laboratory. Trident Compiler. `http://trident.sourceforge.net/`, last accessed: 2008-01-10.

[23] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a Single Assignment Programming Language to Reconfigurable Systems. *Supercomputing*, 21(2):117–130, 2002.

[24] B.A. Draper, A.P.W. Böhm, J. Hammes, W.A. Najjar, J. Ross Beveridge, C. Ross, M. Chawathe, M. Desai, and J. Bins. Compiling SA-C Programs to FPGAs: Performance Results. In *Proc. of the Second International Workshop on Computer Vision Systems (ICVS'01)*, pages 220–235. Springer-Verlag, 2001.

[25] M.B. Gokhale, J.M. Stone, J. Arnold, and M. Kalinowski. Stream-Oriented FPGA Computing in the Streams-C High Level Language. In *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2000)*, 2000.

[26] M.B. Gokhale. Streams-C: Stream-Oriented C Programming for FPGAs. `http://www.streams-c.lanl.gov`, last accessed: 2008-01-08.

[27] Impulse Accelerated Technologies. Impulse C. `http://www.impulsec.com/`, last accessed: 2008-01-07.

[28] S. Vassiliadis, S. Wong, G.N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E.M. Panainte. The MOLEN Polymorphic Processor. *IEEE Transactions on Computers*, 53(11):1363–1375, 2004.

[29] M. Bednara and J. Teich. Synthesis of FPGA Implementations from Loop Algorithms. In *Proc. of the First International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA01)*, pages 1–7, June 2001.

[30] F. Hannig, H. Dutta, and J. Teich. Mapping a Class of Dependence Algorithms to Coarse-Grained Reconfigurable Arrays: Architectural Parameters and Methodology. *International Journal of Embedded Systems*, 2(1/2):114–127, 2006.

[31] G. Snider, B. Shackleford, and R.J. Carter. Attacking the Semantic Gap between Application Programming Languages and Configurable Hardware. In *Proc. of the 2001 ACM/SIGDA 9th International Symposium on Field Programmable Gate Arrays*, pages 115–124, 2001.

[32] ASIM department of Laboratoire d'Informatique de Paris 6 (LIP6). DIgital SYstem Design ENviromenT.
`http://www-asim.lip6.fr/recherche/disydent/`, last accessed: 2008-08-08.

[33] M. Diaby, M. Tuna, J-L. Desbarbieux, and F. Wajsburt. High Level Synthesis Methodology from C to FPGA Used for a Network Protocol Communication. In *Proc. of the 15th IEEE international Workshop on Rapid System Prototyping (RSP'04)*, June 2004.

[34] H. Devos, K. Beyls, M. Christiaens, J. van Campenhout, E.H. D'Hollander, and D. Stroobandt. Finding and Applying Loop Transformations for Generating Optimized FPGA Implementations. *Transactions on High Performance Embedded Architectures and Compilers I*, 4050:159–178, July 2007.

[35] C. Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proc. of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 7–16, September 2004.

[36] D. Ku and G. De Micheli. HardwareC – A Language for Hardware Design (Version 2.0). Technical Report CSL-TR-90-419, Stanford University, August 1990.

[37] G. De Micheli, D. Ku, F. Mailhot, and T. Truong. The Olympus Synthesis System. *IEEE Design & Test*, 7(5):37–53, 1990.

[38] P.L. Flake and S.J. Davidmann. Superlog, a Unified Design Language for System-on-chip. In *Asia and South Pacific Design Automation Conference (ASP-DAC'00)*, 2000.

[39] Open SystemC Initiative. SystemC.
`http://www.systemc.org/`, last accessed: 2008-01-08.

[40] K.D. Nguyen, Z. Sun, P.S. Thiagarajan, and W.F. Wong. Model-driven SoC Design Via Executable UML to SystemC. In *Proc. of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, pages 459–468, December 2004.

[41] Mitrionics. Mitrion Platform. `http://www.mitrionics.com/`, last accessed: 2008-07-07.

[42] SRC Computers. Carte Programming Environment. `http://www.srccomp.com/techpubs/carte.asp`, last accessed: 2008-01-08.

[43] P. Buxa, L. Gorham, M. Lukacs, and D. Caliga. Mapping of a 2D SAR Backprojection Algorithm to an SRC Reconfigurable Computing MAP Processor. In *Proc. of the High Performance Embedded Computing workshop (HPEC 2005)*, September 2005.

[44] M. Geilen and T. Basten. Requirements on the Execution of Kahn Process Networks. In *Proc. of the 12th European Symposium on Programming (ESOP 2003)*, 2003.

[45] S. Meijer, A.C.J. Kienhuis, A. Turjan, and E. de Kock. A Process Splitting Transformation For Kahn Process Networks. In *Proc. of the Design Automation and Test in Europe conference (DATE'07)*, pages 17–19, April 2007.

[46] S. Verdoolaege, H. Nikolov, and T.P. Stefanov. PN: a Tool for Improved Derivation of Process Networks. *EURASIP Journal on Embedded Systems*, 2007.

[47] AT&T Research. Graphviz - Graph Visualization Software. `http://www.graphviz.org/`, last accessed: 2008-06-19.

[48] H. Nikolov, T.P. Stefanov, and E.F. Deprettere. Multi-processor System Design with ESPAM. In *Proc. of the 4th IEEE/ACM/IFIP Int. Conf. on HW/SW Codesign and System Synthesis (CODES-ISSS'06)*, pages 211–216, October 2006.

[49] Synfora inc. *PICO Express – Writing C Applications: Developer's Guide*, 2007.

[50] Y. Tao, H. Nikolov, T.P. Stefanov, and E.F. Deprettere. Heterogeneous Multiprocessor System Design with ESPAM: Integration of Hardware IP Cores. Technical Report 06-21, LIACS, Leiden University, December 2006.

[51] H. Nikolov, T.P. Stefanov, and E.F. Deprettere. Modeling and FPGA Implementation of Applications using Parameterized Process Networks with Non-Static Parameters. In *Proc. of the 13th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 255–263, April 2005.

[52] C. Zissulescu, A. Turjan, A.C.J. Kienhuis, and E.F. Deprettere. Solving Out of Order communication using CAM memory; an implementation. In *Proc. of the 13th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2002)*, November 2002.

[53] A.C.J. Kienhuis, E. Rijpkema, and E.F. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, May 2000.

[54] Xilinx inc. *Fast Simplex Link (FSL) Bus (v2.10a) – Product Specification*, November 2006.

[55] T.J. Shepherd and J.G. McWhirter. *Systolic Adaptive Beamforming – Radar Array Processing*, volume 25 of *Springer Series in Information Sciences*. Springer-Verlag, 1993.

[56] C. Zissulescu, A.C.J. Kienhuis, and E.F. Deprettere. Increasing pipelined IP core utilization in Process Networks using Exploration. In *Proc. of Field-Programmable Logic and Applications (FPL'04)*, pages 690–699, 2004.