Compilation from Matlab to Process Networks Realised in FPGA

Tim Harriss, Richard Walke – QinetiQ Ltd, Malvern, UK
Bart Kienhuis, Ed Deprettere - Leiden University, Leiden, The Netherlands

Abstract

Compaan is a software tool capable of automatically translating nested loop programs, written in Matlab, into parallel Kahn process network descriptions suitable for implementation in hardware. In this paper we present a tool for converting these process networks into FPGA implementations. The QR decomposition algorithm is used to demonstrate the capability of the tool to quickly generate high-performance parallel implementations. This allows us to rapidly explore a range of transformations, such as loop unrolling and skewing, to generate a circuit that meets the requirements of a particular application. We present results showing how the control logic complexity and number of clock cycles vary with these transformations.

1 Introduction

It is now quite common for digital signal processing algorithms to be developed using Matlab, because it has built-in support for many signal-processing operations. Unfortunately, Matlab's imperative model of computation does not directly yield the parallel implementations necessary to address the performance requirements of the signal processing applications we are interested in.

The *Compaan* work, by Leiden University, has shown that a certain class of Matlab programs can be transformed automatically into a parallel process network specification. Such process networks use the *stream-based function* (SBF) model of computation [1]. This model enables easy implementation on an array of DSP chips, or as a custom circuit on an ASIC or FPGA. The latter has become attractive, as the density of FPGAs is sufficient to allow large numbers of processors to be implemented on a single device without the high non-recurring engineering costs associated with producing an ASIC.

The aim of the Compaan environment is to automate the transformation of DSP algorithms specified as nested loop programs (NLP) into parallel hardware implementations. Compaan takes an NLP and uses a three-step process (see Figure 1) to extract parallelism and produce a parallel process network description based on the SBF model of computation. Firstly, the MatParser tool uses data dependence analysis techniques to extract the parallelism from the Matlab code. MatParser produces a single assignment code (SAC) description of the algorithm, in the form of a set of affine nested loops. The next step is DgParser which reduces the data dependencies expressed in the SAC into a *polyhedral reduced dependence graph* (PRDG). Finally, the Panda tool translates the PRDG description into the SBF process network description.

Presented at the 35th Asilomar conference on Signals, Systems, and Computers, November 4 - 7, 2001, Pacific Grove CA USA. ***Copyright © QinetiQ Ltd, 2001***.

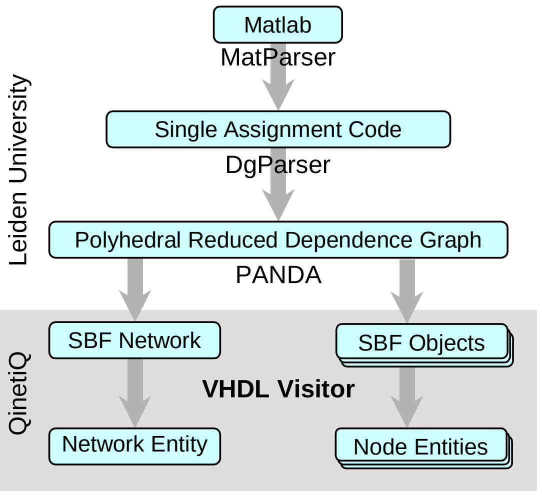

Figure 1: The Compaan process

We have extended the Compaan flow with a fourth step that translates the process networks into hardware. This paper focuses on this fourth step.

The Panda tool constructs an abstract data-structure of the process network using the SBF model of computation. In this data-structure, the SBF network and the various SBF objects or nodes are described. The tool that we have constructed, translates this abstract description into a VHDL network entity and a number of VHDL node entities. The translation is performed by means of a *visitor* [5] - a software engineering technique that makes it easy to operate on the various elements of the abstract data structure generated by Panda.

2 Stream Based Function Computation Model

The SBF computation model describes a number of processing nodes that are interconnected by communication channels, as shown below:

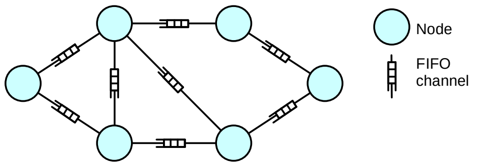

Figure 2: SBF Network Example

The model specifies that the network control is performed locally in the nodes (i.e. the control is distributed), each working independently. The communication channels are defined as infinite FIFO buffers with non-blocking writes and blocking reads, allowing asynchronous communication between the independent nodes. Each SBF node or object is modelled as containing a *controller*, a *state*, and a *function repertoire*, where the latter is F∈ {fa, fb, …fn}.

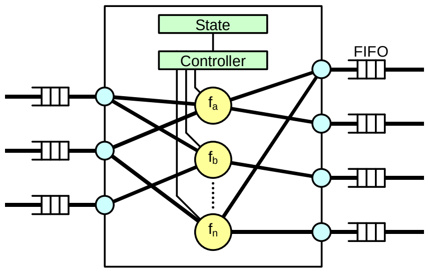

Figure 3: SBF node configuration

The function repertoire defines all the calculations, which can be carried out by a node, and which ports should be used for data input and output for each calculation. The operation of the node is characterised by the order these functions are performed and thus is dictated by the controller. The operation of the controller is described by the *binding function* and the *transition function*. The binding function specifies which repertoire function to use in the current state, and the transition function defines which state follows the current state.

3 Realising Process Networks in Hardware

VHDL was chosen to describe the process network in hardware as it is widely supported by industry standard simulation and synthesis tools. It provides full control over the detail of the implementation and QinetiQ have a library of fixed and floating-point parameterised components optimised for FPGA implementation.

The process network description, in VHDL, is a hierarchical design consisting of a number of SBF node components interconnected at the top level. The FIFOs specified in the process network model of Figure 2 have been absorbed into the nodes.

3.1 Node Design

Each node employs a FIFO buffer on each input port and contains a function unit and a number of multiplexers (mux) and de-multiplexers (demux). This configuration is shown in Figure 4.

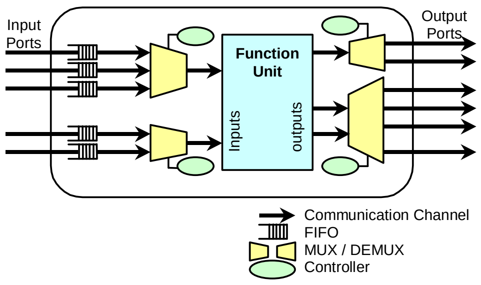

Figure 4: General Node design

In a Compaan process network, the actual calculation performed by each function in a node's repertoire is the same, but the data is read from or written to different ports. This is known as a *function variant repertoire*. The demuxes and muxes provide the variation by switching the data on the inputs and outputs of the function units.

The demuxes are controlled separately to the muxes to allow pipelined function units to empty when the input is stalled. Each controller contains the state, transition function and the relevant part of the binding function defined in the SBF model.

3.2 Communication Channels

Data transfer within the node and between nodes is performed over communication channels employing a request-acknowledge hand-shaking protocol. Hand-shaking is necessary because the FIFOs and function units may not always be ready to transmit valid data.

3.3 FIFO Design

The process network model specifies that communications take place through unbounded FIFO channels. Fortunately there is a bound at which a particular algorithm will run deadlock free. At present this is found through simulation.

As shown in Figure 5, the FIFO design is based around a dual port RAM with counters to generate the read and write addresses. Additional control is required to check for empty and full states and to interface with the communication channels.

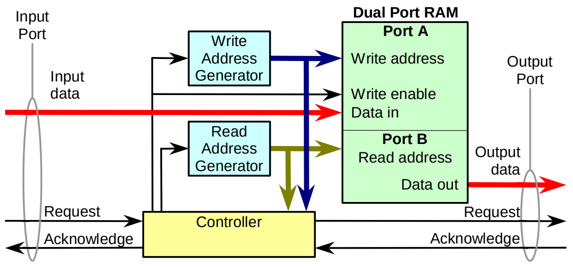

Figure 5: FIFO design

4 An Example: QR Decomposition

QR decomposition is a mathematical algorithm commonly used to solve an over-specified set of linear equations in a least squares sense. Our interest in this algorithm is for calculating weights in an adaptive beamforming system [2]. The fact that it falls under the umbrella of nested loop algorithms, as well as being well understood, has made it a good test for the Compaan tool flow throughout development. It also makes it a good test for the complete algorithm to hardware flow discussed in this paper.

QR decomposes a matrix **X,** using unitary rotations, into an upper-triangular matrix **R**, which in our application can be back substituted to provide the least squares weights. The QR algorithm employed is based around the iterative Givens Rotations method [2].

As shown in Figure 6, QR employs two operations: *vectorize* and *rotate*. Vectorize takes a vector formed by an element of **X** and an element of **R** and rotates it through an angle such that the **X** element is forced to zero. The rotate operation takes a similar vector but rotates it through an angle previously calculated by a vectorize operation. Within the QR implementation, these two operations are combined to rotate a row of the **X** matrix vector against each row of the **R** matrix, zeroing the leading element in the **X** row each time.

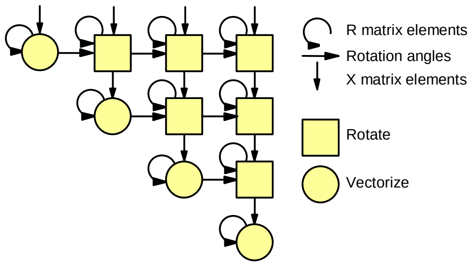

Figure 6: QR data dependencies

For purposes of explanation, part of the Matlab code written as input to Compaan is shown below, and describes the Givens rotations calculations.

```
for k = 1:1:K,
   for j = 1:1:N,
      [r(j,j),t] = vectorize( r(j,j),x(k,j) );
      for i = j+1:1:N,
         [r(j,i),x(k,i),t] = rotate(r(j,i),x(k,i),t);
      end
   end
end
```

In addition, other loops may be required to initialise the **R** matrix and provide input and output for the **X** and **R** data matrices. In the description, three loop indices have been used: j counts down the rows of the **R** matrix, i counts along each row of **R** and k counts complete Givens rotations updates (i.e. rows of **X**). The loop bounds K and N are constant parameters whose values are the number of QR updates and number of columns in the **X** matrix respectively.

The process network produced by Compaan for this code consists of five interconnected nodes; as shown in Figure 7.

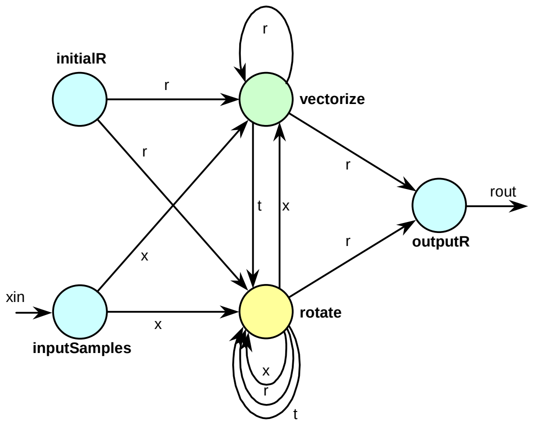

Figure 7: QR Process Network

The calculation is performed by the vectorize nodes and rotate nodes. The other three handle all the initialisation and input/output. It is worth noting that the network contains one node for each function call in the original code (note that the input/output function calls were omitted from the code segment presented earlier in the paper). The detail of the calculation performed by a node, in order to execute its function, is as yet undefined, and is not relevant at this level of the design process since Compaan only addresses data flow. Ultimately, the node detail must be provided by the designer although there is no reason why it cannot be another Compaan process network. We consider a specific implementation of these nodes later in this paper.

The VHDL description contains the network interconnection and control required to schedule the data through the nodes. It also implements the parameters K and N as generics, and hence can be synthesised for any size of QR problem.

5 A Basic Implementation of QR

In order to implement QR on an FPGA, the five function units were developed. The vectorize and rotate nodes were designed to make use of the built-in fixed-point multipliers available in the Xilinx Virtex II FPGA family [3]. The basic vectorize algorithm was implemented using adders, multipliers and a multiplier based inverse-square-root core. The function units were developed for 16-bit fixed-point input and output data.

A simulation of this complete design was performed and the results compared against a set of previously generated reference values. Figure 8 shows the simulation waveforms for a system with 20 inputs and 40 input sample vectors (N=20, K=40).

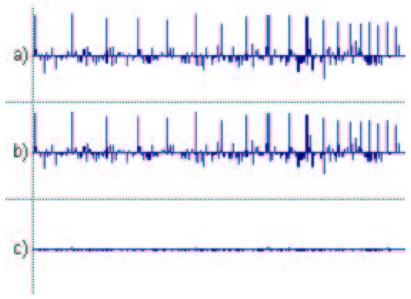

Figure 8: QR simulation results - a) reference, b) network output, c) error.

The error between the actual and reference signals is of the order of 2% and can be attributed to the difference between double precision floating-point and 16-bit fixed-point arithmetic.

The VHDL network and node descriptions were converted to a circuit netlist using a commercial tool, and then passed through Xilinx's FPGA place-and-route tools. The resulting layout is shown in Figure 9.

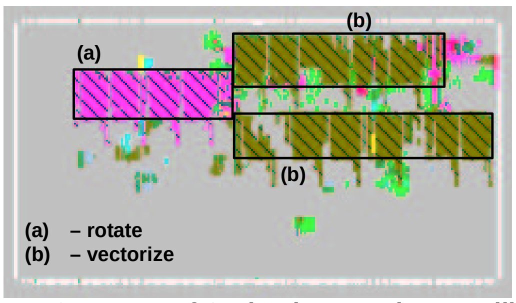

Figure 9: Layout of QR implementation on Xilinx XCV1000E FPGA.

The large, dense areas correspond to the vectorize and rotate function units, and the remaining resources make up the control structures.

The XCV1000E device was chosen to allow implementation on current hardware. Because of this, it was necessary to use multiplier cores instead of the dedicated multiplier resources available on Virtex II. The cores correspond to the 15 dense rectangular areas on the layout, which would not be necessary on Virtex II, leaving a relatively small amount of control logic.

The data extracted from the simulation and place-and-route is as shown in the following table:

	Total clock cycles	Max clock freq. (MHz)	Updates per milli-second	Total LUTs used for node control	Node control LUTs per update per micro-second
QR	2300	35	15.2	4978	327.2

Table 1: Design properties for QR; N=20, K=40.

A single update consists of 20 vectorize operations and 190 rotate operations, each of which contain 11 or 4 multiplies respectively and 2 adds. This gives a total of 980 multiply and 420 add operations per update which equates to over 20 million operations per second, compared to a potential 595 million. This poor utilisation can be attributed to two things: Firstly, limitations in the control and FIFO designs; Work is ongoing to address these. Secondly, and perhaps more significantly, limitations in the schedule, causing processors to stall. Addressing the latter is considered in the next section.

It is worth noting that it took about 15 minutes to process the design from Matlab to a hardware implementation suitable for download onto an FPGA. The majority of this time is taken up with synthesis and place-and-route – Compaan took about 10 seconds to process the Matlab file.

6 Comparison of Optimising Transforms

The first transform, called *skewing*, can be used to address the schedule limitations mentioned in the previous section. Inefficient use of pipelined function units occurs when the data dependencies in a network are such that a node is forced to wait for input data, possibly from one of its own outputs. This is the case for the QR network where the vectorize node waits for data from the rotate node (see Figure 6). The transformation skews the order of execution so that nodes are operating on data from previous updates (iterations of k loop).

The second transformation performs loop *unrolling*, forcing Compaan to duplicate the contents of the loop a number of times. This results in more nodes, thus increasing parallelism. Unrolling can be used in conjunction with skewing to optimise a design for specific performance and resource requirements. Both unrolling and skewing can be achieved by making simple modifications to the input code. This is currently done automatically using a Perl script but ultimately could be included as an optional transform in Compaan.

To compare the effect of these transformations, simulations were run on the QR algorithm with various combinations of the transformations. We are particularly interested in the control complexity so, for the purpose of the tests, the function units were replaced with simple single cycle components. Consequently, the data given is only useful for comparing the transformations and not the QR implementation. The tests were carried out with an input data matrix size of 10 by 6. For the unrolled algorithms, the outer loop of the main calculation was unrolled by a factor three. This has the effect of creating three of everything in that loop and partitioning the operations between them. Consequently, three vectorize and three rotate nodes were created.

Table 2 shows values for the operating time and speed, as well as resource usage, and a ratio for revealing the resource cost of the speed increase gained using the transformations. The resource usage is measured in LUTs (Look Up Tables) - a primitive component on the Virtex family of FPGAs [3]. One Virtex CLB contains four LUTs. It can be seen from these figures that unfolding improves the throughput of the design, but skewing only has a positive effect when combined with unrolling. The drop in clock frequency is due to the increased complexity of the node control logic.

	Total clock cycles	Max clock freq. (MHz)	Updates per micro-second	Total LUTs used for node control	Node control LUTs per update per micro-second
Basic	851	91.1	1.07	270	252
Skew	761	70.7	0.93	315	339
Unroll	341	70.0	2.05	886	431
Skew + Unroll	276	68.2	2.47	1362	550

Table 2: Comparison of designs with various transformations.

The time taken for these designs to be processed from Matlab to a hardware implementation ranged up to 60 minutes. The majority of the time was spent in synthesis and place-and-route.

7 Conclusions and Future Work

The tests discussed in this paper demonstrate the ability of the tool to automatically generate synthesisable VHDL implementation of an algorithm specified in Matlab. The fact this process takes minutes rather than months, significantly reduces the design time of a system over a VHDL-only design flow.

We have also illustrated the use of network transformations for design optimisation. These transformations can be made automatically and provide the potential for extensive design space explorations to be undertaken quickly. These can either be performed automatically as a batch job and the best picked, or the designer can use their knowledge and experience to refine the design. Since the optimisations can be applied automatically at a high level means the designer does not need to worry about the complex low-level designs and can, thus, concentrate on higher-level issues.

To increase the range of potential applications, the visitor requires many refinements to improve the VHDL implementation. Currently, the control forms a bottleneck for the clock speed in complex designs; there is a lot of scope for optimisation such as pipelining. Another area that needs improvement is the communications throughout the system. Currently, three cycles are required for a single transaction. This significantly affects the performance by limiting the maximum potential data rate into a function unit to one sample every three cycles.

This work clearly demonstrates that SBF process networks translate well into hardware. More generally, it shows how distributed control can produce efficient implementations. Although only a single FPGA is targeted in the examples, this distributed control can be used to target systems with multiple FPGAs and microprocessors.

8 Acknowledgements

This work has been sponsored by the MoD Corporate Research Programme.

9 References

[1] B. Kienhuis, E. Deprettere, E. Rypkema, "Compilation from Matlab to Process Networks", Cases ’99, October 1-3, Washington 1999.

[2] T. J. Shepherd and J. G. McWhirter, “Systolic Adaptive Beamforming – Radar Array Processing”, Springer Series in Information Sciences, Vol. 25, 1993, Springer-Verlang Berlin.

[3] Xilinx Inc., “Virtex-II 1.5V Field-Programmable Gate Arrays”, DS031 (v1.2) January 15, 2000

[4] T. Stefanov, E. Deprettere, B. Kienhuis, “Exploring Application Model Instances in System-Level Design”. Progress Workshop, 2001.

[5] Gamma et al., “Design Patterns, Elements of Reusable Object-Oriented Software”, Addison-Wesley Professional Computing Series, 1994