Compilation from Matlab to Process Networks Realized in FPGA

Tim Harriss and Richard Walke
QinetiQ Ltd, Malvern, UK

Bart Kienhuis and Ed Deprettere
Leiden University (LIACS), Leiden, The Netherlands

April, 2002

Abstract. Compaan is a software tool that is capable of automatically translating nested loop programs, written in Matlab, into parallel process network descriptions suitable for implementation in hardware. In this article, we show a methodology and tool to convert these process networks into FPGA implementations. We will show that we can in principle obtain high performing realizations in a fraction of the design time currently employed to realize a parameterized implementation. 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. The QR decomposition algorithm is used to demonstrate the capability of the tool. We present results showing how the number of clock cycles and calculations-per-second vary with these transformations using a simple implementation of the function units. We also provide an indication of what we expect to achieve in the near future once the tools are completed and applied the transformations to parallel, highly pipelined implementations of the function units.

Keywords: Process Networks, Stream-based Model of computation, Nested Loop Programs, Matlab, FPGA, Skewing, Unrolling.

1. Introduction

With the drive for mobility, it is a common requirement for computing systems to handle high bandwidth data signals, such as video, and transmit them, possibly in real-time, via wireless and wired connections. The processing needed to compress, protect, and transmit data is demanding and requirements for real-time performance and low power consumption motivate the use of optimized hardware implementations. Fortunately, the repetitive and regular nature of these signals results in the adoption of a data-flow model of computation which is very amenable to a hardware implementation.

Data-flow processing has been commonplace within the military systems application domain for many years. Sensor processing within real-time applications, such as sonar, radar, and communications, has demanded levels of processing that in the past has required ASIC implementations or even multiple board implementations. More recently, FPGA technology has provided a configurable alternative. We have been exploiting this technology, in common with the commercial communications sector, to provide digital receivers and beamformers for military systems. The regular and repetitive operations, low-wordlength, and fixed-point arithmetic found in these applications, make the architecture of FPGAs particularly well suited.

Many digital signal processing applications are conveniently written in the Matlab programming language. It is, however, very hard to obtain straight from a Matlab program a good implementation onto an FPGA that is time efficient (Haldar et al., 2001). The contribution of this article is that it shows that by changing the model of computation of the Matlab program, which is sequential, into a model of computation that is closer to data-flow models, i.e. process networks, a time efficient implementation is possible on FPGAs. These implementation can be obtained quickly and furthermore, this article shows that high-level optimization like unrolling and skewing can be applied directly to the Matlab program to further improve implementations.

1.1. Floating-point QR decomposition implementation

We have been experimenting with the use of configurable computing, in the form of FPGA technology, to implement weight calculation in adaptive systems. Specifically, we have implemented floating-point arithmetic on FPGA and used it to construct a processor for performing QR decomposition from which a wide range of sensor array algorithms can be implemented. QR decomposition is employed, as opposed to other techniques, to minimize the arithmetic dynamic range requirements (Shepherd and McWhirter, 1993). Substantial savings in operator size are obtained by reducing the wordlength of the mantissa and exponent of our floating-point numbers to that which is just sufficient to meet the accuracy requirements of our systems. The level of optimization possibly depends on the application, but it has been found that relatively low wordlengths are possible, and using 14-bit mantissa we obtain in excess of 20 GigaFLOPS of computation on a single FPGA (Walke et al., 1999).

The benefits of floating-point over fixed-point in FPGA are algorithm and application dependent. For our applications, greater performance can be obtained using QR decomposition with floating-point arithmetic. The dynamic range provided by the exponent allows us to use a fast square-root free algorithm, employing a reduced number of operations with lower wordlength arithmetic compared to fixed-point arithmetic. Furthermore, the floating-point exponent and mantissa can easily be increased to single[1] or higher precision, should that be required by the application. Although FPGA architectures are currently not optimized for performing floating-point arithmetic, their gate-count is now so high that they offer a solution in our application with performance that is substantially higher than those based on programmable processors.

[1] which is defined by the IEEE as 24-bit mantissa and 8-bit exponent

1.2. Design time of a parameterized QR array processor

The design time of an FPGA implementation has been a serious issue. The transformations to obtain our architecture have had to be derived manually. Furthermore, the scheduling and linear systolic processor implementation that was produced, had to be described manually in a fashion that was parameterized. This took a single person almost a full year.

The intention is now to extend this design so that the size of the problem, specifically, the number of inputs, can be programmable at run-time. To achieve this in the current design, the implementation must be regenerated. Whilst this is acceptable within a lab, it is currently an unacceptable practice within deployed systems. Although, in principle, a customer could recompile the core for a new problem size, there are issues of distributing source code, providing synthesis and physical tools on location, and supporting the activity. With regard to the latter, it is too time-consuming to test fully all possible combinations of parameters. Consequently, it is not possible to guarantee a successful build of any size, particularly, as the process is vulnerable to the failure of the implementation tools.

We have two choices to generate a software programmable core. Either manually redesign the core to be software programmable, or develop a method to generate such an implementation automatically. The former is time consuming, very specific to the problem, and difficult to get correct for all parameter values. The latter is fast, correct by construction, and generic, which is strategically important as QR decomposition is the first of many requirements.

1.3. Compaan: a solution

Compaan is a set of tools for translating an algorithm expressed as a set of nested for-loops within an imperative language such as Matlab into a description based on process networks (Kienhuis et al., 2000). The schedule of each process is parameterized in terms of the original loop dimensions. As a result, a software programmable implementation can be generated automatically. The processes will read on start-up the parameter values that are stored in some registers, allowing a network to resize at run-time. Moreover, the process networks employ buffered communication, which allows us to partition processors across multiple devices without considering, in too much detail, the precise latency of a path.

Compaan takes an affine nested loop program (NLP) (Held, 1996) and uses a three-step flow (see Figure 1) to extract parallelism and produce a parallel process network description based on the *Stream Based Functions* (SBF) model of computation (MoC) (Kienhuis and Deprettere, 2001). Firstly, the array dataflow analysis compiler *MatParser* (Kienhuis, 2000) extracts all inherent parallelism from the Matlab code and produces a single assignment code (SAC) description of the algorithm in the form of a set of affine nested loops. Next, *DgParser* reduces the data dependencies expressed in the SAC into a polyhedral reduced dependence graph (PRDG). Finally, the third program,*Panda*, translates the PRDG description into the SBF process network description. We have extended the Compaan flow with a fourth step that trans-

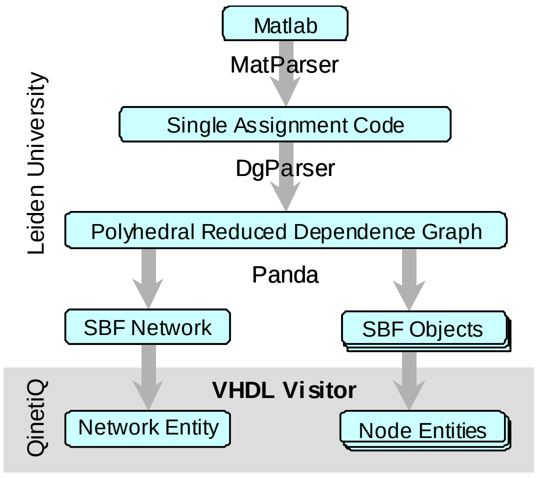


Figure 1. The Compaan flow with the extension (as highlighted) that is discussed in this article that deals with the translation of process networks written in the SBF model of computation into hardware mapped on an FPGA.

lates the process networks into hardware (highlighted section in Figure 1). This article focuses on this fourth step. The Panda tool constructs an abstract data-structure of the process network using the SBF model of computation. Our new tool 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 (Gamma et al., 1994) - a software engineering technique that makes it easy to operate on the various elements of the abstract data structure generated by Panda.

The remainder of this article is structured as follows. In Section 2, we discuss the MoC employed. Its architectural implementation in hardware is considered in Section 3. As an example, QR decomposition is used to demonstrate our methodology, and is briefly discuss in Section 4. The results of a direct implementation of this example are presented in Section 5 using highly pipelined functions. Since the resulting level of throughput is very poor, improving transformations are described in Section 6 and applied using simple functions in Section 7. This section demonstrates the effectiveness of these transformations and indicates the level of performance we expect from a single FPGA once our methodology is complete. We conclude this article in Section 8.

2. Stream-based Function Computation Model

The process network that is derived by Compaan is specified using the SBF model of computation. This model is more general than any one of the determinate dataflow models like homogeneous dataflow or synchronous dataflow. It is as general as the Kahn process network model (Kahn, 1974) but has process behavior that is more structured than general process models are. It is a natural model for specifying stream based applications at a high level of abstraction.

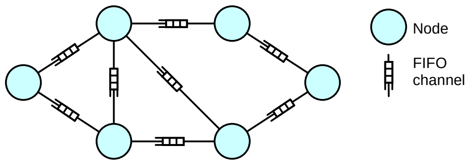


Figure 2. An SBF Network.

Stream-based applications are described as a *network* of SBF objects communicating concurrently with each other using channels, as shown in Figure 2. The essential components are *Stream-Based Function Objects* and *Channels*. These channels are defined as infinite FIFO buffers and buffer possibly unbounded streams of tokens communicated between a producing SBF object and a consuming SBF object. An SBF object accesses channels with non-blocking writes and blocking reads, allowing asynchronous communication between the independent nodes, which gives an SBF network its deterministic behavior (Kahn, 1974).

An SBF object has an inside view and an outside view. Inside an SBF object, the following three components are present as shown in Figure 3: a *set of functions*, a *controller*, and a *state*. The set of functions is referred to as the *function repertoire* of an SBF object and in Figure 3 consists of the set $\{f_a, \dots, f_n\}$. At the outside, an SBF object exposes read and write *ports*. These ports connect to channels, allowing SBF objects to communicate streams with each other.

Each repertoire function knows from which ports it should read and to which ports it should write to. All repertoire functions together define the calculations that can be carried out by a node. However, the actual behavior of the node is characterized by the order in which the functions of the repertoire are executed. This order is dictated by the controller in terms of a binding and transition function. The binding function specifies which repertoire function to use in the current state and the transition function defines the next state from the current state.

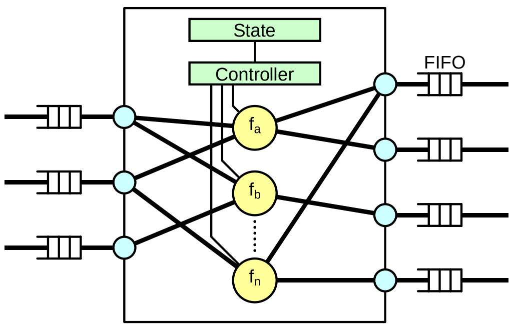


Figure 3. An SBF Object

3. Realizing Process Networks in Hardware

Our philosophy is that an easy mapping can only take place if the model of computation matches the model of the architecture (Kienhuis et al., 2002). FPGAs are ideal for implementing distributed control structures due to the inherent parallelism available in FPGAs. However, the model of computation of Matlab does not fit the parallelism available on the FPGA. As a consequence, deriving an time efficient – as oppose to hardware efficient – implementation from the Matlab will be very hard and time consuming, although it is tried (Haldar et al., 2001).

In Matlab, the for-loops impose sequential ordering and the matrices used in the program are stored in some global memory, for example, the memory of a PC. On the other hand, the SBF model of computation describes an algorithm in terms of distributed control and memory. Each node in Figure 2 executes a control sequence as dictated by the controller, independently of other nodes and synchronizes with others using blocking reads. The data flowing through the network is stored in the distributed FIFO buffers implementing the channels. In conclusion, the computational model of the process network description is very different from the computational model of the Matlab description which it is derived from.

It is the distributed control and memory of an SBF network that simplifies the transition from a high abstract model of computation to a hardware implementation on an FPGA. In this section, we discuss our architectural model and indicate how the SBF objects map to this model. Because the MoC of the SBF network matches this model of architecture, the mapping is straightforward. This differentiates this work, for example, from high-level synthesis that would take the control-dataflow graph (CDFG) (De Micheli, 1994) of the original Matlab program as their model of computation. The CDFG model doesn't deal effectively with task-level parallelism. Because the model of computation matches our architectural model, the network description derived by Compaan can be directly converted into a this hardware model

(See Figure 1). In that sense, Compaan can be seen as a compiler that converts the sequential model of computation of Matlab into a process network model of computation that is more amenable to the model of architecture mapped onto FPGAs.

3.1. Model of Architecture

The model of architecture used for realizing a process network in hardware is shown in Figure 4. It consists of two controllers, a number of input and output switches, a function unit and some additional interfacing (glue) logic (not considered further in this article). The FIFO buffers have been included in the node to simplify the network level connections.

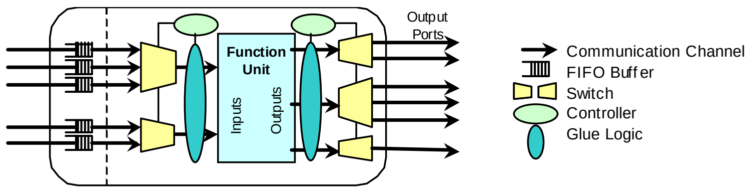


Figure 4. Hardware node model.

The function repertoire is not implemented as a number of functions, but is instead mapped onto the function unit. After all, only one repertoire function will execute at a time. We see the function unit as a programmable element for which the instruction set is defined by the function repertoire. To get the right data to and from the function unit, a number of switches are needed that route data to and from channels.

Although in general the repertoire can contain many different functions, the repertoire derived by Compaan currently consists of the same function but with different connections to input and output ports. The same function with different input/output connections is called a function variant (Kienhuis and Deprettere, 2001). The function unit implements a single function, and the function variant lead to multiplexers around it.

The control described by the binding and transition function are realized in a node using nested if-statements and for-loops containing linear expressions for the conditions and loop limits. They can easily be translated to logic. The linear expressions can be implemented using arithmetic circuits, and the decisions (i.e., how to set the switches for the right function variant) are made using multiplexers driven by the results of the linear expression calculations. Additional logic monitors the handshaking signals and when a data transfer has occurred, i.e., a function has executed on the function unit, so triggering the controller to update the state.

Compaan produces a network description containing a specification of the control, but does not specify the content of the function units. To Compaan, the functions are black boxes that need to be implemented by a designer. It will often be most effective to implement the function units as pipelines.

Pipelining enables increased throughput but introduces latency, which must be accommodated within the schedule and the implementation of the controller. Therefore, the architectural model shows two identical controllers: one at the input and one at the output. A consequence of pipelining is that the output data is delayed from the input data depending on the number of pipeline stages used. Therefore, the input and output controller could work on different states. For example, with a full pipeline of depth 10, whilst the output switches are routing data associated with state 4, the input switches will be routing data associated with state 14. Moreover, the difference between the two states may not be constant if the input is stalled due to lack of data or the output destination is full. Providing separate input and output controllers allows the input and output switches to behave and operate independently.

The SBF process network model specifies that communications take place through unbounded FIFO channels. In reality, the FIFO buffers must be bounded. Through simulation, the bounds at which a particular algorithm will run deadlock free can be determined (Parks, 1995). Yet, it is possible that a FIFO can become full and a source node is unable to output further data, forcing it to stall. It is also possible that a stalled node is unable to accept input data from a channel even if it is available on other channels the node is connected to. To accommodate such situations, all the communication in the network use an available/request handshaking protocol.

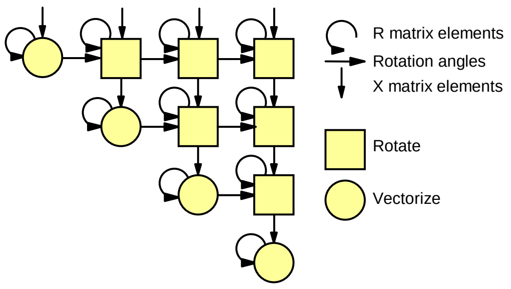


Figure 5. QR Data Dependencies.

4. An Example: QR Decomposition

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 (Shepherd and McWhirter, 1993).

```
%parameter N 1 16;
%parameter T 1 1000;
for k = 1 : 1 : T,
  for j = 1 : 1 : N,
    for i = j : 1 : N,
      if i <= j,
        [r(j,i),angle] = vec(r(j,i),x(k,i));
      else
        [r(j,i),x(k,i)] = rot(r(j,i),x(k,i),angle);
      end
    end
  end
end
```

Figure 6. The QR algorithm in Matlab Code.

Figure 5 shows the data dependencies for QR. Two operations are employed: 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.

The Matlab code used by Compaan, is shown in Figure 6 and describes a simple form of the Givens rotations calculations. For clarity, the loops required to initialize the R matrix and read in the X matrix have not been shown. Also, the loops required to obtain the resulting R data matrix is not shown. 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 T and N are parameters whose values are the number of QR updates and number of columns in the X matrix respectively. For these parameters a range of acceptable values is given (i.e., using the `%parameter` keyword).

The process network produced by Compaan for this code consists of five interconnected nodes as shown in Figure 7. The calculation is performed by the vectorize nodes and rotate nodes. The other three handle all the initialization and input/output. It is worth noting that the network contains one node for each function call in the original Matlab code. 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. The designer must provide the node detail. There is no reason why it cannot be another Compaan process network. We consider a specific implementation of these nodes later in this article.

The VHDL produced by the visitor program contains the network interconnection and description of the nodes. Currently, the parameters T and N are defined as VHDL generics, and the system can be synthesized for any problem size. Alternatively, the parameters can be specified as inputs to enable run-time resizing of the problem.

5. A Basic Implementation of QR

This section describes a real implementation of the QR decomposition algorithm on an FPGA. The implementation is the direct realization of the process network generated by Compaan and shown in Figure 7. The function units were developed manually. We used the Squared Givens Rotation algorithm, which is square-root free and employs fewer operations in the rotate functions than conventional square-root algorithm. However, higher dynamic range arithmetic is required. This can be accommodated by floating-point arithmetic. Although as a rule, floating-point is relatively expensive to implement in FPGA, the mantissa size requirements are less than the wordlength of fixed-point arithmetic. Furthermore, the reduced number of operations results in floating-point arithmetic offering a significant overall saving over a fixed-point implementation (Walke, 1997). Also, there is the additional benefit of not having to scale input data. QinetiQ's Quixilica library of floating-point cores have been used. These are optimized for Xilinx Virtex families of FPGA (QinetiQ Ltd, 2001). The floating-point units are highly pipelined

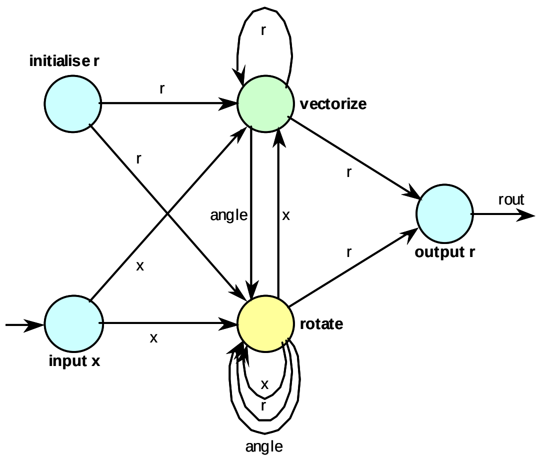


Figure 7. QR Process Network.

and resulted in latencies of 41 and 61 for the rotate and vectorize units respectively.

The VHDL network and node descriptions were converted to a circuit netlist using Synplify, a commercial circuit synthesis tool, and then passed through place-and-route tools from Xilinx. VHDL simulation yielded the number of cycles taken to complete program execution and the place-and-route tool provided the maximum achievable clock frequency for the system. This data is summarized in Table I. The utilization of the rotate and vectorize

Table I. Results for the real QR implementation

Parameter Values	Total	Operations		Utilization (%)		Clock	Performance
N=5, T=15	Cycles	Vec	Rot	Vec	Rot	MHz	Mega FLOPS
Basic QR	6937	75	150	1.09	2.18	27	12.6

functions is calculated as the percentage of cycles in which data entered the functional unit. The vectorize and rotate functional units contain 11 and 16 floating point operations [2] respectively. Therefore, the full simulation contains (75*11) + (150*16) = 3225 floating point operations. These operations were executed in 6937 cycles leading to 3225/6937 = 0.46 floating point operations per cycle. With a maximum clock frequency of 27MHz, this provides a computation rate of 27*0.46=12.6 million floating point operations per second (MegaFLOPS).

The low computational rate is due to the low unit utilization and clock frequency. The low utilization can be addressed using particular transformations and will be shown in Section 6 and 7. A utilization rate approaching 100% for the rotate and 50% for the vectorize unit can be achieved. From timing analysis of the design, the low clock frequency is due to long combinatorial logic paths in the controller implementation. However, pipelining the controller design will improve the situation greatly and clock speeds of 100MHz should be achievable.

6. Optimization Transforms

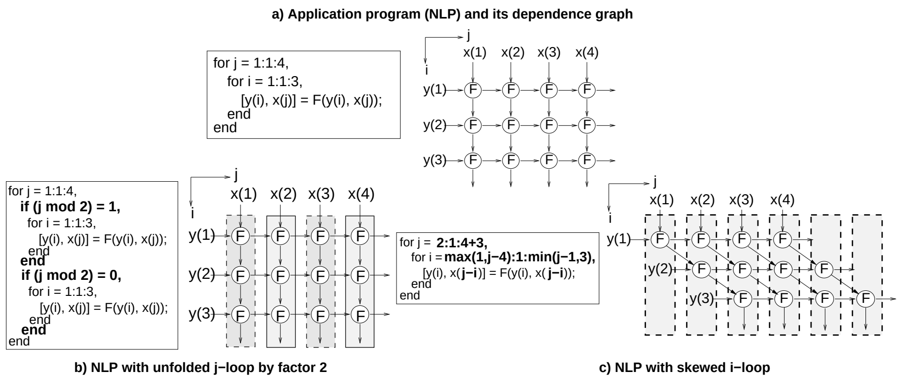
The Compaan tool chain results in a unique process network for a given algorithm in Matlab. The unique process network for the QR algorithm is given in Figure 7. However, this process network is unlikely to be of use to the designer. As shown in Table I, a direct implementation results in low utilization and computation rate. Therefore, we present in this section two

[2] Add, subtract, multiply, or divide

algorithmic transformations, i.e., unrolling and skewing, that change the behavior of the derived process network, but not the functionality. The changed behavior leads to better implementations.

The transformations take as input an NLP and a set of parameters. The output of the unrolling transformation is an NLP, which is functionally equivalent to the input program but with enhanced task-level parallelism. The skewing transformation makes the potential parallelism in the input NLP explicit. We have implemented these transformations in a toolbox called *Mat-Transform* (Stefanov et al., 2002), which operates directly on the NLP source code without using some intermediate representation like dependence graphs, signal-flow graphs, or data-flow graphs corresponding to the NLP.

Nevertheless, to explain unrolling and skewing, we will use a dependence graph (DG, see Figure 8) as it represents a graphical representation of an NLP. The nodes in the DG represent the NLP functions that are executed in each loop-iteration and the edges represent the data dependencies between the functions.


a) Application program (NLP) and its dependence graph
j
i
x(1)
x(2)
x(3)
x(4)
y(1)
y(2)
y(3)
F
b) NLP with unfolded j−loop by factor 2
c) NLP with skewed i−loop

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

```
for j = 1:1:4,
   if (j mod 2) = 1,
     for i = 1:1:3,
       [y(i), x(j)] = F(y(i), x(j));
     end
   end
   if (j mod 2) = 0,
     for i = 1:1:3,
       [y(i), x(j)] = F(y(i), x(j));
     end
   end
end
```

```
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
```

Figure 8. The unrolling and skewing transformations.

6.1. Unrolling and Skewing

The transformation of *unrolling* is to create a new NLP with enhanced task-level parallelism. For example, the NLP in Figure 8-a has two loops (with iterators j, i), which can be unrolled. Figure 8-b shows the NLP in which the j-loop is unrolled by a factor of 2. The two pieces of code bounded by the ”if” statements in Figure 8-b have become mutually exclusive. This characteristic is exploited by MatParser to partition the program into two processes (tasks) that can operate in parallel. Unlike common approaches, in which either the loop control is removed through loop unrolling (Muchnick, 1997) or the DG is folded (Parhi, 1999), our approach to get the desired degree of parallelism - at the task level - is to copy a loop body a number of times in such a way that these copies are mutually exclusive.

The transformation of *skewing* is to create a new NLP in which the bounds of the loops and the indices of the variables are changed to make the potential parallelism in the original NLP explicit. For example, skewing the i-loop of the program in Figure 8-a leads to the NLP in Figure 8-c. The effect of our skewing transformation is visualized by the dependence graph (DG) in Figure 8-c. This DG explicitly shows that the nodes inside a dashed box can be executed in parallel because there are no data dependences between these nodes. This property can again be exploited by MatParser to partition the program into processes that run in parallel. Moreover, inside these processes code can be executed effectively in a pipeline fashion because there are no data-dependencies between nodes that could cause stalls.

7. Assessment of Optimizing Transforms

In comparing the transforms, we are particularly interested in the number of cycles taken to complete a simulation. So for the purpose of these comparisons, the function units have been implemented as simple delays with single cycle latency. Again, we use the algorithm shown in Section 4. The results are based on an implementation that takes fifteen input vectors, each containing five values (i.e. N=5, T=15). The remainder of this section will describe what is obtained from each transformation, and then present and discuss the results.

7.1. The Effect of Skewing on QR

The main calculations of the QR problem are performed in the rotate and vectorize function units in the nodes of Figure 7. To make best use of the hardware resources, both these units should be fully utilized. As can be seen from the data dependence graph (See Figure 5), there are a greater number of rotate operations than vectorize operations and, thus, the utilization of the vectorize unit must be less than that of the rotate unit. For the case of the problem size specified above (N=5), an update contains 10 rotate and 5 vectorize operations, indicating that the vectorize unit is expected to have a maximum possible utilization of half the utilization of the rotate unit. Therefore, the best we can aim for is 100% utilization of the rotate unit and 50% utilization of the vectorize unit.

A direct implementation of the QR process network results in very inefficient utilization of the function units (see Table I); nodes stall whilst waiting for results of the preceding operations. As explained in the previous section, skewing should increase the utilization of the function units by re-ordering the operations such that data is available when needed. Skewing can be performed both vertically (j axis) and horizontally (i axis) to give the following four possibilities:

a.) Basic (non-transformed)
b.) i-axis skewed
c.) j-axis skewed
d.) i and j axes skewed

For each of these possibilities, we have generated and simulated a VHDL implementation and found the results as shown in Table II and Table III. [!hbt] Table II contains results for the complete simulation run, and Table III

Table II. Results of skewing QR for the complete simulation

Operation	Total	Operations		Utilization (%)	
N=5, T=15	Cycles	Vec	Rot	Vec	Rot
Basic	456	75	150	16.45	32.89
Skew j by 1	344	75	150	21.80	43.60
Skew i by 1	206	75	150	36.41	72.82
Skew i and j by 1	171	75	150	43.86	87.72

contains results for just a single update, i.e., not within the run-in and run-out periods. As can be seen, the run-in and run-out periods account for a large difference between the two tables. With a larger value for T (number of input vectors), the overall utilization shown in Table II would become nearer to those in Table III. Note that the utilization of the Rotate and Vectorize functions is higher then the utilization given in Table I because the functions are not pipelined.

To understand the effect of skewing, it is necessary to consider these direct data dependencies and the order of the operations. Figure 9 shows the critical direct dependencies for the four QR implementations. The critical dependencies define the number of cycles taken to complete one update. Note that the majority of the critical data dependencies exist between operations on the same row.

Consider Figure 9-a; the direct data dependencies along each row mean that each iteration of the rotate node must wait for the previous one to com-

Table III. Results of skewing QR for a single normal update

Operation	Total	Operations		Utilization (%)	
N=5, T=15	Cycles	Vec	Rot	Vec	Rot
Basic	30	5	10	16.67	33.33
Skew j by 1	22	5	10	22.73	45.45
Skew i by 1	12	5	10	41.67	83.33
Skew i and j by 1	10	5	10	50.00	100.00

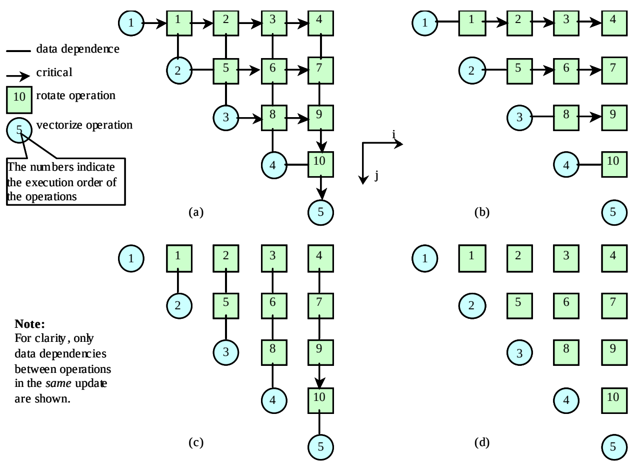

data dependence
critical
rotate operation
vectorize operation
The numbers indicate the execution order of the operations
Note: For clarity, only data dependencies between operations in the *same* update are shown.
i
j
(a)
(b)
(c)
(d)

Figure 9. Critical direct data dependencies in QR with (a) no skew, (b) j axis skewed, (c) i axis skewed, (d) i and j axis skewed.

plete. In addition, on the first row, the vectorize operation must be completed before the first rotate operation can start. Each of the critical data dependencies introduces a two cycle delay (the latency of the FIFO). The nine critical dependencies in addition to the ten rotate operations, give twenty-eight cycles. A further two cycles are introduced by the first and last vectorize operations, providing a total of thirty cycles to complete one update:

$$(9 \times 2) + 10 + 2 = 30 \ \text{ cycles}. \tag{1}$$

This can be generalized to (note: only valid for N>2):

$$\text{Cycles \ per update for basic QR} = 2\left(\frac{(N-1)(N-2)}{2} + 3\right) + \left(\frac{N(N-1)}{2}\right) + 2, \tag{2}$$

where N is the number of elements in the input vectors.

Skewing the j axis effectively removes the vertical critical dependencies, as shown in Figure 9-b, and the two cycle delays. Another effect is that the first and last vectorize operations do not affect the number of cycles since they can now be performed in parallel with the rotate operations. The number of cycles is now given by:

$$\text{Cycles \ per update for QR with j skewed} = 2\left(\frac{(N-1)(N-2)}{2}\right) + \left(\frac{N(N-1)}{2}\right). \tag{3}$$

Likewise, skewing the i axis removes the horizontal critical dependencies. Skewing i has a greater effect than skewing j because the majority of critical data dependencies are horizontal (See Figure 9-c). The following equation gives the number of cycles per update:

$$\text{Cycles per update for QR with i skewed} = 2\left(1\right)+\left(\frac{N(N-1)}{2}\right). \tag{4}$$

Finally, combining these two yields the best results by removing all the critical data dependencies (see Figure 9-d). For this, the number of cycles for one normal update is simply the number of rotate operations, providing the 100% utilization as shown in Table III.

7.2. THE EFFECT OF UNROLLING ON QR

In hardware, greater throughput can be gained by introducing spatial parallelism into a system. With Compaan, this can be achieved by applying the unrolling transformation to one or more of the loops in the algorithm. Here, we demonstrate this by unrolling the k loop of our QR algorithm for varying factors. This has the effect of generating process networks with more than one vectorize and rotate operation and scheduling different updates on different pairs of vectorize and rotate nodes. With a greater number of function units performing calculations, we should expect to see the number of cycles fall at approximately the reciprocal of the unroll factor. For example, with an unroll factor of two, we should see a halving of the number of cycles taken to complete the simulation. The simulations were performed for fifteen input

Table IV. Results of unrolling QR for the complete simulation

Operation	Total	Operations		Utilization (%)	
N=5, T=15	Cycles	Vec	Rot	Vec	Rot
Unroll k by 1	456	75	150	16.45	32.89
Unroll k by 2	246	75	150	15.24	30.49
Unroll k by 3	166	75	150	15.06	30.12
Unroll k by 4	136	75	150	13.79	27.57
Unroll k by 5	119	75	150	12.61	25.21

vectors, each containing five samples (i.e., T=15, N=5) and with a function unit latency of one. Table IV shows the results for unrolling factors one to five where a factor one is the same as non-transformed. Only the results for the complete simulation are included here since the utilization of the units over a single normal update are the same as shown in the first row of Table III.

From the table we can see that as expected, increasing the unroll factor reduces the number of cycles taken to complete the simulation. Figure 10 shows a plot of the actual number of cycles taken compared with a 1/X trend. The difference between the two curves is due firstly to input and output periods similar to those in Section 7.1 and, secondly, to run-in and run-out periods introduced by the R dependencies between updates of the QR algorithm. The second of these causes a node scheduled to perform an operation in one update to wait for the corresponding operation in the previous update. For example, with an unroll factor of two, the odd updates are scheduled onto nodes 3 and 4, and the even scheduled onto 5 and 6. In order for node 5 to perform its first operation, node 3 must have finished its first operations and produced the R value. This effect would be more dramatic if larger latency function units were used.

The size of the disparity between the two curves grows with the unroll factor due to an increase in the relative significance of the of run-in and run-out cycles to the total number of cycles.

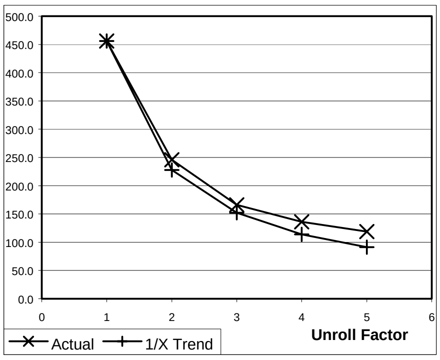


Figure 10. Cycles for simulation.

7.3. THE EFFECT OF UNROLLING AND SKEWING ON QR

The skewing transformation allows manipulation of the scheduling to provide maximum utilization of pipelined function units. Unrolling provides the designer with the ability to vary the amount of parallelism in a system. Typically, a designer will want to combine unrolling and skewing to create the optimal implementation for the system requirements. To demonstrate this, a number of simulations were performed for algorithms with the j axis skewed and the k loop unrolled for varying factors and the results are shown in Table V. By combining the two transformations, we would expect to achieve similar function unit utilization as with just the j axis skewed but with in-

Table V. Results of unrolling and skewing QR for the complete simulation

	Total	Operations		Utilization (%)	
Operation	Cycles	Vec	Rot	Vec	Rot
Skew j by 1 and unroll k by 1	344	75	150	21.80	43.60
Skew j by 1 and unroll k by 2	184	75	150	20.38	40.76
Skew j by 1 and unroll k by 3	142	75	150	17.61	35.21
Skew j by 1 and unroll k by 4	118	75	150	15.89	31.78
Skew j by 1 and unroll k by 5	113	75	150	13.27	26.55

creased spatial parallelism, thus reducing the overall number of cycles taken to complete.

As in Section 7.2, the number of cycles decreases as the unroll factor increases. As before, the function unit utilization for a single normal update is the same for all unroll factors, but overall, the utilization drops. This drop is due to the combined run-in and run-out periods from skewing and unrolling as described in previous sub-sections.

It should be noted that unrolling and skewing will make the control more complex, requiring more time to update the control state and more logic resources to implement. These effects are not considered in this article as the control logic in our application is small relative to the functional units and the extra controller time can be accommodated with greater depth of pipelining without loss in clock rate. The additional latency of this pipelining in the controller will only have a very minor impact on the total number of clock cycles for a QR decomposition.

7.4. Estimation of Skew and Unroll Results for the Floating Point QR Implementation

As demonstrated in Section 5, a direct implementation of the process network for QR decomposition is extremely inefficient. On the other hand, in this section it has been shown how the throughput of an implementation can be drastically improved using transformations. The results have been presented for a simple processor with a latency of 1 cycle (a total latency of 3 cycles including the FIFOs). Nevertheless, the much higher latencies of the actual function implementations can be accommodated by applying greater levels of skew. This is work currently being undertaken and we do not yet have implementation results. However, we can provide the reader with an estimate of the expected level of performance to be achieved for a single FPGA using this route when complete.

Single rotate and vectorize functions were used in the QR decomposition process network and the nature of the problem is such that the vectorize function was under-utilized. For the presented case of N=5 the vectorize function is used half as often as the rotate function. For N=20, the utilization of the vectorize drops to 10.5%. To improve utilization, we can reduce the throughput of the vectorize function and save resources. If the vectorize function is implemented using a single multiplier and adder, then both components will have a similar level of utilization for the case of N=16. The vectorize function will take longer, but is used less frequently and thus is active for the same amount of time as the rotate function. For N above 16, the utilization of the vectorize function will drop further, but this will have little effect on the overall utilization, as it employs only 2 operators whereas the rotate function employs 16. On the other hand, an interesting alternative to redesigning the implementation of the vectorize function may be to only unroll the rotate function (i.e., obtain multiple rotate functions for one vectorize function). Using the proposed design flow it will be easy to perform such design exploration.

The total size of a vectorize and rotate function pair is approximately 3,000 slices (where a slice is a basic unit of Xilinx Virtex FPGAs). This represents 7.8% of an XC2V6000 - currently the largest FPGA in production. If an unroll factor of 10 were used, then 78% of the chip would be utilized. As discussed in Section 5, a pipelined controller is expected to run at a clock rate in excess of 100MHz on current FPGA devices. Hence, a total computation rate of 18 GigaFLOPS would be achieved from a single FPGA running at 100MHz. This is close to what has been achieved manually and in a fraction of time compared to the manual design.

It should be noted that this estimate assumes that the number of ’run-in’ and ’run-out’ cycles is minimized. For high skew factors and small arrays, the number of these cycles can be very large relative to the intervening period over which the array is fully occupied. However, in our applications, the QR decomposition is performed repeatedly, so we can arrange for one problem to be ’run-in’ as another is ’run-out’. In Compaan this can be achieved by employing a further outer loop that repeatedly executes the QR decomposition function.

8. Conclusions and Future Work

In this article, we have presented a tool flow which will enable us to take an NLP written in Matlab and implement it automatically in FPGA. This translation process takes minutes rather than months, and so significantly reduces the design time of a system over a VHDL-only design flow. Currently, it takes a single person a full year to design a parameterized implementation of QR onto an FPGA. As shown in this article, Compaan has the potential to reduce the design time of QR to days or even minutes should the exploration be simple. Even though we have not implemented in FPGA a skewed and unrolled version of QR using the pipelined vectorize and rotate units, we have shown that we can perform the skewing and unrolling necessary for an efficient implementation.

We illustrated the use of network transformations for design optimization. These transformations are parameterized and can be made automatically, providing 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. Alternatively, the designer can use their knowledge and experience to refine the design. Since the optimizations can be applied automatically at a high level means the designer does not need to worry about the complex low-level designs and thus can concentrate on the higher-level issues.

To increase the range of potential applications, the Visitor program requires many refinements to further improve the VHDL implementation. Currently, the control forms a bottleneck for the clock speed but a generic architecture for the controller has been designed so that it can be easily pipelined.

This work also demonstrates that the SBF model of computation, used to express the process networks translate well into hardware. There is a clear correspondence between the elements of the SBF Object and the model of architecture. Because the semantic model of the SBF process network and the hardware model is the same, we know what to expect at a very high level of abstraction. We also showed how distributed control can produce efficient implementations. Although only a single FPGA is currently targeted in the examples, this distributed control can be further exploited to target systems with multiple FPGAs and microprocessors.

9. Acknowledgments

The development of the VHDL visitor has been sponsored by the United Kingdom Ministry of Defence Corporate Research Programme. The MatTransform has been sponsored by the Program for Research on Embedded Systems & Software (Progress) program of STW, the Netherlands, contract AES5021. The Compaan tool chain has been establish by various research efforts done at University of California at Berkeley, Technical University Delft, and the University of Leiden, LIACS.

References

De Micheli, G.: 1994, *Synthesis and Optimization of Digital Circuits.* McGraw-Hill International Editions.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides: 1994, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series.

Haldar, M., A. Nayak, A. Choudhary, and P. Banerjee: 2001, 'A System for Synthesizing optimized FPGA hardware from Matlab'. In: *Proceedings of IEEE international conference on Computer Aided Design ICCAD'2001*. San Jose, USA, pp. 314 – 319.

Held, P.: 1996, 'Functional Design of Data-Flow Networks'. PhD thesis, Delft University of Technology, The Netherlands.

Kahn, G.: 1974, 'The Semantics of a Simple Language For Parallel Programming'. In: *Proc. of the IFIP Congress 74*. North-Holland Publishing Co.

Kienhuis, B.: 2000, 'MatParser: An array dataflow analysis compiler'. Technical report, University of California at Berkeley. UCB/ERL M00/9.

Kienhuis, B., E. Deprettere, P. van der Wolf, and K. Vissers: 2002, *A Methodology to Design Programmable Embedded Systems*, Vol. 2268 of *LNCS*, pp. 18 – 37. Springer Verlag.

Kienhuis, B. and E. F. Deprettere: 2001, 'Modeling Stream-Based Applications using the SBF model of computation'. In: *Proceedings of IEEE workshop on Signal Processing Systems (SIPS'2000)*. Antwerp, Belgium.

Kienhuis, B., E. Rijpkema, and E. F. Deprettere: 2000, 'Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures'. In: *8th International Workshop on Hardware/Software Codesign (CODES'2000)*. San Diego, USA.

Muchnick, S.: 1997, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc.

Parhi, K.: 1999, *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, Inc.

Parks, T.: 1995, 'Bounded Scheduling of Process Networks'. Ph.D. thesis, University of California at Berkeley.

QinetiQ Ltd: 2001, 'Quixilica Floating-Point FPGA Cores'. http://www.quixilica.com/pdf/qx_fpl.pdf.

Shepherd, T. and J. McWhirter: 1993, 'Systolic Adaptive Beamforming – Radar Array Processing'. In: *Springer Series in Information Sciences*, Vol. 25. Berlin: Springer-Verlag.

Stefanov, T., B. Kienhuis, and E. Deprettere: 2002, 'Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances'. In: *Proceedings of 10th International Symposium on Hardware/Software Codesign*. Colorado, USA.

Walke, R. L.: 1997, 'High-Sample Rate Givens Rotations for Recursive Least Squares'. Ph.D. thesis, University of Warwick.

Walke, R. L., R. W. M. Smith, and G. Lightbody: 1999, '20GFLOPS QR processor on a Xilinx Virtex-E FPGA'. In: *proceedings of SPIE advanced signal*.