Accelerator for Event-based Failure Prediction

Acceleration of an Extended Forward Algorithm for Failure Prediction on FPGA

Master's Thesis submitted to the

Faculty of Informatics of the *Università della Svizzera Italiana*

in partial fulfillment of the requirements for the degree of

Master of Science in Informatics

Embedded Systems Design

presented by

Simon Maurer

under the supervision of

Prof. Miroslaw Malek

Janaury 2014

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Simon Maurer
Lugano, 29. Janaury 2014

Abstract

Acknowledgements

Contents

Figures

Tables

Listings

Chapter 1

Introduction

In today's live it becomes increasingly important, that computer systems are dependable. The reason being, that computer systems are used more and more in areas where the failure of such a system can lead to catastrophic events. Banking, public transportation and medical engineering are only a few examples of areas employing large and extremely complex systems. The increasing complexity of computer systems has a direct impact on their maintainability and testability. It is simply impossible to guarantee that a piece of software comes without any faults. On top of that, the same problematic arises with the hardware components which also may contain faulty parts but also get increasingly prone to failures due to decay of material.

In the event of a system failure it is of course desirable to fix the system as soon as possible in order to minimize the downtime of the system (maximize the availability). This can be accomplished by using different types of recovery techniques, e.g. Check-pointing (create checkpoints to roll back/forward), system replication (switch to a redundant system), fail over (reboot). All these techniques require a certain amount of time to complete the recovery process, time that is very expensive. In order to minimize this time, techniques have been developed to anticipate upcoming failures. Such a technique is described in [24].

The work presents a new algorithm to predict failures and compares the results with other techniques. The accuracy of the presented algorithm to predict failures proves to be better compared to the other techniques, has however the drawback of increased complexity and hence increased computation time. It is very important to keep the computation overhead very low in order to maximize the time between the prediction of a failure and the actual event of the failure. One way to decrease the computation time is to design a hardware accelerator for the prediction algorithm. The design of such an accelerator is outlined in this document.

1.1 Problem Statement

1.2 Motivation

The email of Felix left some doubts to whether the acceleration of the algorithm is useful. The following list will give some arguments to justify the work.

Too many parameters to be identified, estimated and set
Considering an embedded system, this is usually not a problem because the parameters

are defined during the design phase and will never be changed afterwards.

Limited performance scalability
There are studies available claiming otherwise. The discussion of Neumanns work will provide some arguments against this statement.

Industry trends point towards cloud
In embedded systems it will still be beneficial to predict failures of single nodes. It is however important to keep the power and computational footprint low. This will be one of the major challenges. On the other hand, I think it would also be possible to also use this algorithm to monitor a distributed system and predict failures. It is only a matter of getting defining the events to feed to the algorithm.

1.3 Structure

Chapter 2

State of the Art

This section provides an overview of the state of the art in the different fields of research that are relevant for the thesis. This includes failure prediction methods, existing solutions to accelerate failure prediction algorithms and acceleration techniques in general.

2.1 Failure Prediction

A very detailed overview of failure prediction methods is given in [25]. The survey discusses i.a. the techniques used as comparison in the main reference [17, 16, 28, 9] as well as the technique described in the main reference [24].

More recent work uses hardware counters of a general purpose CPU and combines them with software instrumentation to analyze failures of single processes (e.g grep, flex, sed) [32]. As industry heads more and more towards cloud computing, it has been proposed to use information of interaction between nodes (instead of analyzing single nodes) in order to analyze and predict failures of a distributed system [26, 21].

2.2 Accelerator

The main goal of this master thesis is to accelerate an adaptation of the forward algorithm. Proposals for a GPU based accelerator for the classic forward algorithm are described in [20, 18]. Further, several proposals to accelerate the Viterbi algorithm (which is closely related to the forward algorithm) have been published: [2] presents an architecture for a lightweight Viterbi accelerator designed for an embedded processor datapath, [11, 19, 22] describe a FPGA based accelerator for protein sequence HHM search and [29] describes i.a. an approach to accelerate the Viterbi algorithm from the HMMER library using GPUs.

Focusing on a more general approach for acceleration, [13] proposes an FPGA implementation of a parallel floating point accumulation and [31] describes the implementation of a vector processor on FPGA.

Quite some research has been done on the question what type of technology should be used to accelerate certain algorithms: [5] presents a performance study of different applications accelerated on a multicore CPU, on a GPU and on a FPGA, [12] discusses the suitability of FPGA and GPU acceleration for high productivity computing systems (HPCS) without focusing on a

specific application and [15] also focuses on HPCS but uses the Basic Linear Algebra Subroutines (BLAS) as comparison and also takes CPUs into account.

It may be interesting to also think about an acceleration of the model training. Similar work has been done by accelerating SVMs (Support Vector Machines): [4] describes a FPGA based accelerator for the SVM-SMO (support vector machine - sequential minimal optimization) algorithm used in the domain of machine learning and [1] proposes a new algorithm and its implementation on a FPGA for SVMs.

Chapter 3

Event-based Failure Prediction

describe complete concept in a brief overview: non-failure model, failure models, classification

This section provides a brief overview of the computational steps done by the proposed algorithm [24].

brief description of the idea behind the algorithm, HSMM, Events, etc

To be able to understand the formal expression of the algorithm, first a definition of the used parameters is provided.

- N: number of states
- M: number of observation symbols
- L: observation sequence length
- R: number of cumulative probability distributions (kernels)

The delay of the event at time t_k with respect to the event at time t_{k-1} is described as

$$d_k = t_k - t_{k-1} \tag{3.1}$$

3.1 Data Processing

3.2 Training of the Model

One part of the algorithm is the model training. This part is not described here. The features to be trained by the model training are however important in this context because they are used by the adapted forward algorithm. Following the features:

- π_i, forming the initial state probability vector π of size N
- $b_i(o_j)$, forming the emission probability matrix B of size $N \times M$
- p_{ij}, forming the matrix of limiting transmission probabilities P of size $N \times N$
- $\omega_{ij,r}$, the weights of the kernel r
- $\theta_{ij,r}$, the parameters of the kernel r

3.3 Sequence Processing

The following description will provide a complete blueprint of the extended forward algorithm, that allows to implement it, but without any explanations or proofs related to the formulation. The adapted forward algorithm is defined as follows:

$$\alpha_0(i) = \pi_i b_{s_i}(O_0) \tag{3.2}$$

$$\alpha_k(j) = \sum_{i=1}^{N} \alpha_{k-1}(i) v_{ij}(d_k) b_{s_j}(O_k); \quad 1 \leq k \leq L \tag{3.3}$$

where

$$v_{ij}(d_k) = \begin{cases} p_{ij} d_{ij}(d_k) & \text{if } j \neq i \\ 1 - \sum_{\substack{h=1 \\ h \neq i}}^{N} p_{ih} d_{ih}(d_k) & \text{if } j = i \end{cases} \tag{3.4}$$

with

$$d_{ij}(d_k) = \sum_{r=1}^{R} \omega_{ij,r} \kappa_{ij,r}(d_k | \theta_{ij,r}) \tag{3.5}$$

forming the matrix of cumulative transition duration distribution functions $D(d_k)$ of size $N \times N \times L$.

For simplification reasons, only one kernel is used. Due to this, the kernel weights can be ignored. Equation 3.5 can then be simplified:

$$d_{ij}(d_k) = \kappa_{ij}(d_k | \theta_{ij}) \tag{3.6}$$

Choosing the Gaussian cumulative distribution results in the kernel parameters μ_{ij} and σ_{ij}:

$$\kappa_{ij,gauss}(d_k | \mu_{ij}, \sigma_{ij}) = \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{d_k - \mu_{ij}}{\sqrt{2}\sigma_{ij}}\right)\right] \tag{3.7}$$

explain difference to the non-extended forward algorithm and introduce a standard notation for the transition probabilities (eg v: extended a: basic, tp: general) stick to this in the whole document

The last set of forward variables α_L are then summed up to compute a probabilistic measure for the similarity of the observed sequence compared to the sequences in the training data set. This is called the sequence likelihood:

$$P(\boldsymbol{o} | \lambda) = \sum_{i=1}^{N} \alpha_L(i) \tag{3.8}$$

where $\lambda = \{\pi, P, B, D(d_k)\}$.

To prevent α from going to zero very fast, at each step of the forward algorithm a scaling is performed:

$$\alpha_k(i) = c_k \alpha_k(i) \tag{3.9}$$

with

$$c_k = \frac{1}{\sum_{i=1}^{N} \alpha_k(i)} \tag{3.10}$$

By applying scaling, instead of the sequence likelihood (equation 3.8), the sequence log-likelihood must be computed:

$$\log(P(\boldsymbol{o}|\lambda)) = -\sum_{k=1}^{L} \log c_k \tag{3.11}$$

where $\lambda = \{\pi, P, B, D(d_k)\}$.

3.4 Classification

explain classification

and finally the classification is performed:

$$\text{class}(s) = F \iff \max_{i=1}^{u} \left[\log P(\boldsymbol{s}|\lambda_i) \right] - \log P(\boldsymbol{s}|\lambda_0) > \log \theta \tag{3.12}$$

with

$$\theta = \frac{(r_{\bar{F}F} - r_{\bar{F}\bar{F}})P(c_{\bar{F}})}{(r_{F\bar{F}} - r_{FF})P(c_F)} \tag{3.13}$$

classification without scaling

Multi-class classification without scaling?

To calculate θ, the following parameters need to be set:

- $P(c_{\bar{F}})$: prior of non-failure class
- $P(c_F)$: prior of failure class
- $r_{\bar{F}\bar{F}}$: true negative prediction
- r_{FF}: true positive prediction
- $r_{\bar{F}F}$: false positive prediction
- $r_{F\bar{F}}$: false negative prediction

3.5 Metrics

cite results from Felix

Chapter 4

Theoretical Analysis of the Forward Algorithm

This chapter provides details about the forward algorithm and available (and useful) parallelization techniques applicable on the algorithm. The generally known forward algorithm as well as the extended version proposed in the reference work is discussed. Further, scaling techniques of the forward variables and their impact on data representation choices are presented. Finally the observations presented in an overview and an appropriate choice on possible acceleration hardware is made.

4.1 Serial Implementation and Complexity

The sequential implementation of the basic forward algorithm is represented in listing 4.1. It consist of three parts: the initialization step, the computation of consecutive forward variables and the final step, where the likelihood is computed. The initial α variable is computed by multiplying the initial state probability with the emission probability of the first observation symbol of the sequence (cf equation 3.2). The computation of the following forward variables consists of three nested loops: the outer loop iterates over the L sets of N α variables, where each variable depends on the prior computed α variable and the k-th observation symbol of a sequence. The first inner loop iterates over the N α variables of one set, where each variable is computed with the most inner loop. The two nested inner loops form the Matrix-Vector-Vector multiplication

$$\alpha_{k+1} = TP * \alpha_k \cdot B(o_k) \tag{4.1}$$

where α_k is a vector of size N of the prior computed α variables, TP a matrix of size $N \times N$ containing the transition probabilities and $B(o_k)$ a vector of size N containing the emission probabilities of the k-th observation symbol. Note that the first multiplication is a Matrix-Vector multiplication that results in a vector, which is then multiplied element-wise with the vector $B(o_k)$. The equation 3.3 describes the formal definition of the forward algorithm. In the last step the likelihood is computed, by summing up all elements of the last forward variable α_L (cf equation 3.8).

As proposed by the reference work, the forward variables can be scaled, in order to prevent the result from getting very small due to the continuous multiplication of probabilities. The

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% computation of the forward algorithm without scaling
%
% @param N:             number of states
% @param L:             number of observation symbols
% @param PI:            initial state probability vector. size N
% @param B:             matrix of emission probabilities. size N, L
% @param TP:            transistion probabilities. size N, N
% @param oL:            indices of all observed symbols. size 1, L
% @return Ps:           probability likelihood
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [Ps] = forward_s_basic(N, L, PI, B, TP, oL)
    % initialize forward variables
    for i=1:N,
        alpha(i) = PI(i)*B(i, oL(1));
    end
    % forward algorithm
    for k=2:L,
        for j=1:N,
            alpha_new(j) = 0;
            for i=1:N,
                alpha_new(j) += alpha(i) * TP(i, j);
            end
            alpha_new(j) *= B(j, oL(k));
        end
        alpha = alpha_new;
    end
    % compute likelihood
    Ps = 0;
    for i=1:N,
        Ps += alpha_new(i);
    end
end
```

Listing 4.1. Forward Algorithm

implementation of the proposed scaling method is shown in listing 4.2. The scaling is formally defined by the equations 3.9 and 3.10. Due to the scaling, instead of the likelihood, the log-likelihood is computed. Equation 3.11 gives the formal definition.

The algorithm to compute the sequence likelihood proposed by the reference work is an extension to the forward algorithm presented in the listings 4.1 and 4.2. Instead of constant transition probabilities, the extended algorithm computes a new transition probability matrix (size $N \times N$) for each arriving observation symbol, by considering the delay of the new symbol with respect to the prior symbol. The computation of the transition probability matrix TP is implemented with listing 4.3 and formally defined by the equations 4.5, 3.5, 3.6 and 3.7. As described in chapter 3.3, also here for reasons of simplification, only one kernel is used. In the sample code the Gaussian cumulative distribution function is used. The function needs to be called for every k.

The order of time complexity of the algorithm is $O(LN^2)$. While the complexity increases with the introduction of scaling and/or the extension, the order of complexity stays the same. The same is true for the space complexity which is of the order $O(N^2)$.

4.2 Parallelism and Faisable Parallelization

By applying parallelization methods one aims to increase the throughput or reduce the latency of a task, or to achieve both at the same time. This can be done at the cost of increased usage of parallel computation units, memory and memory bandwidth.

Considering only the basic forward algorithm (listing 4.1), the computation of the likelihood is divided in $L+1$ steps: the initialization, $L-1$ identical intermediate steps and the finalization. Because of the recursive nature of the algorithm, all steps (except the initialization) depend on the previously computed forward variables. For this reason a direct parallelization of the steps is not possible. However, at every arrival of a new observation symbol, the last L elements of the observation symbol sequence are used to compute the likelihood (cf. figure 4.1). This can be exploited to pipeline the steps in order to increase the throughput. By building a pipeline of $L+1$ stages, where each step of the forward algorithm corresponds to a pipeline stage, a likelihood is computed at every completion of a step, with a latency of $(L+1) * t_{step_{max}}$, where $t_{step_{max}}$ is the time needed to complete the computation of the most complex step (each stage of the pipeline must take the same amount of clock cycles). The throughput of a pipelined compared to a non-pipelined system is increased by factor L (assuming an infinite runtime or by ignoring the setup time). Another and more important fact, that makes the pipeline architecture very beneficial in this particular case: the configuration allows to load the transition probabilities TP and the emission probabilities $b_i(o_k)$ for all steps at the same time, which reduces the load operations by factor L. This is visualized by the table 4.1. The table shows the pipeline stages with input values that are fed to the stage before the execution (note, that the input values TP and B always depend on the same observation symbol. The parameter d_k of the transition probabilities can be ignored in this case, because only in the extended forward algorithm the they depend on d_k. This will be discussed further when the extension is considered) and the output values resulting after the execution of the pipeline stage. Figure 4.2 shows a schematic representation of the pipeline.

By considering all dissimilar steps of the forward algorithm, more parallelization options can be found. In the initial step, N components of the first forward variable α_1 are computed by multiplying independent pairs of an initial state probability π_i and an emission probability of the

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% computation of the forward algorithm with scaling
%
% @param N:             number of states
% @param L:             number of observation symbols
% @param PI:            initial state probability vector. size N
% @param B:             matrix of emission probabilities. size N, L
% @param TP:            transistion probabilities. size N, N
% @param oL:            indices of all observed symbols. size 1, L
% @return Ps:           probability likelihood
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [lPs] = forward_s_scaling(N, L, PI, B, TP, oL)
    % initialize forward variables
    for i=1:N,
        alpha(i) = PI(i)*B(i, oL(1));
    end
    % scaling
    alpha_sum = 0;
    for i=1:N,
        alpha_sum += alpha(i);
    end
    scale_coeff(1) = 1 / alpha_sum;
    for i=1:N,
        alpha(i) *= scale_coeff(1);
    end
    % forward algorithm
    for k=2:L,
        for j=1:N,
            alpha_new(j) = 0;
            for i=1:N,
                alpha_new(j) += alpha(i) * TP(i, j);
            end
            alpha_new(j) *= B(j, oL(k));
        end
        % scaling
        alpha_sum = 0;
        for i=1:N,
            alpha_sum += alpha_new(i);
        end
        scale_coeff(k) = 1 / alpha_sum;
        for i=1:N,
            alpha_new(i) *= scale_coeff(k);
        end
        alpha = alpha_new;
    end
    % compute log likelihood
    lPs = 0;
    for i=1:L,
        lPs -= log(scale_coeff(i));
    end
end
```

Listing 4.2. Forward Algorithm with scaling

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% computation of the extended transition probabilities
%
% @param N:         number of states
% @param dk:        delay of k-th observation symbol
% @param cdf_param: parameters for the cdf
% @return v:        extended transition probabilities
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [v] = compute_tp(N, dk, cdf_param)
    % compute all elements of v
    for i=1:N,
        for j=1:N,
            v(i, j) = normcdf(dk, cdf_param.mu(i, j), cdf_param.sigma(i, j));
        end
    end
    % correct diagonal elemnts of v
    for i=1:N,
        for j=1:N,
            v_sum(i) += v(i, j);
        end
    end
    for i=1:N,
        v_sum(i) -= v(i, i);
        v(i, i) = 1 - v_sum(i);
    end
end
```

Listing 4.3. Extension of the Forward Algorithm with only one kernel (Gaussian)

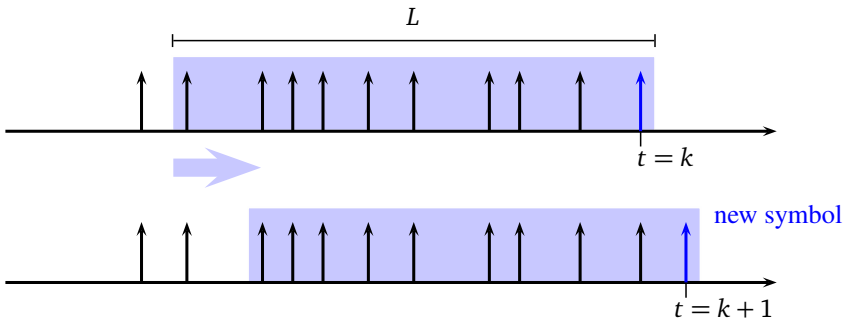

Figure 4.1. Sliding window over an observation sequence of the last $L = 10$ observation symbols

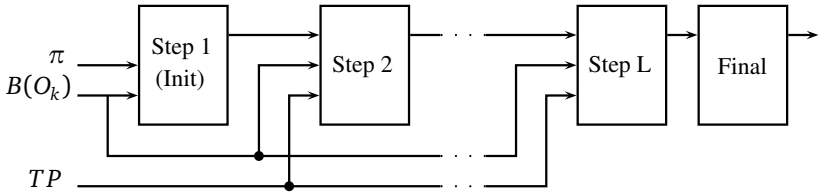

Figure 4.2. Pipelined Forward Algorithm

		Pipeline				
Symb	I/O	Init	Step 2	...	Step L	Final
O_1	in	$B(O_1)$	$B(O_1)$, $TP(d_1)$, 0	...	$B(O_1)$, $TP(d_1)$, 0	0
	out	$\alpha_1(O_1)$	0	...	0	0
O_2	in	$B(O_2)$	$B(O_2)$, $TP(d_2)$, $\alpha_1(O_1)$	...	$B(O_2)$, $TP(d_2)$, 0	0
	out	$\alpha_1(O_2)$	$\alpha_2(O_{1,2})$	...	0	0
$\vdots$		$\vdots$	$\vdots$		$\vdots$	$\vdots$
O_L	in	$B(O_L)$	$B(O_L)$, $TP(d_L)$, $\alpha_1(O_{L-1})$	...	$B(O_L)$, $TP(d_L)$, $\alpha_{L-1}(O_{1,\dots,L-1})$	0
	out	$\alpha_1(O_L)$	$\alpha_2(O_{L-1,L})$	...	$\alpha_L(O_{1,\dots,L})$	0
O_{L+1}	in	$B(O_{L+1})$	$B(O_{L+1})$, $TP(d_{L+1})$, $\alpha_1(O_L)$	...	$B(O_{L+1})$, $TP(d_{L+1})$, $\alpha_{L-1}(O_{2,\dots,L})$	$\alpha_L(O_{1,\dots,L})$
	out	$\alpha_1(O_{L+1})$	$\alpha_2(O_{L,L+1})$	...	$\alpha_L(O_{2,\dots,L+1})$	$Ps(O_{1,\dots,L})$
$\vdots$		$\vdots$	$\vdots$		$\vdots$	$\vdots$

Table 4.1. Pipelined Forward Algorithm, with observation symbol O_k and its delay d_k. Here $O_{i,\dots,k}$ is a short notation for $O_i, \dots, O_k$

first observation symbol $b_i(o_0)$. This can be fully parallelized by replicating the multiplication operation N times. Doing this results in a increase of the throughput by factor N and a decrease of the latency by factor $\frac{1}{N}$, assuming that N multipliers are available and the memory bandwidth is able to provide a data throughput N times higher than in the sequential case (including the memory interface).

The computation of the following forward variables α_k, with $k = 2 \dots L$ are similar. To compute the N elements of one step, the Matrix-Vector-Vector multiplication described by equation 4.1 must be performed. Considering first only the Matrix-Vector multiplication, this can be parallelized by decomposing the matrix in to subsets and then use multiple computational units to perform multiplication and/or accumulation operations in parallel on the subsets. An intuitive decomposition can be done either by block-striped matrix partitioning (decomposition into subsets of rows or columns) or by checkerboard block matrix partitioning (decomposition in rectangular sets of elements). These partitioning methods are shown in figure 4.3. The number of subsets must correspond to the number of available computational units to perform the necessary operations. The choice of decomposition is heavily dependant on the accelerator architecture (e.g. communication between computational units, memory architecture). The resulting vector can then be multiplied element wise by the emission probability vector, which is again the same case as described above. In case of the block-striped matrix partitioning, the maximal achievable increase of the throughput is a factor of N and the latency can be decreased by factor $\frac{1}{N}$, assuming that N computational units are available to perform the multiplication and accumulation operation on each subset, N multipliers to compute the final element wise vector-vector multiplication and a memory interface, that can handle a data throughput that is N times higher than in the sequential case. The checkerboard partitioning yields a lower gain but may be considered in case of fixed computational unit architecture (CPU, GPU) in order to increase the utility of recourses available in each unit. Apart from homogeneous partitioning methods as mentioned above also inhomogeneous solutions have been proposed [7, 6]. These are not considered in this work, as the focus lies on homogeneous computation units.

Further parallelization can be done by using a reduction tree to accumulate the elements in the matrix-vector multiplication process. Instead of using one computation unit and accumulate the values sequentially, N units can be used to first multiply two operands together, then

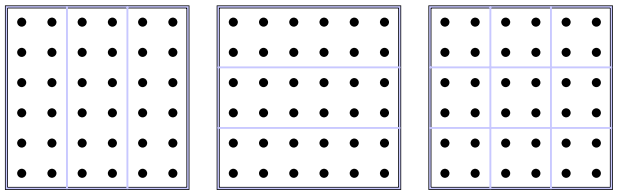

Figure 4.3. Matrix partitioning (from left to right): column-block-striped, row-block-striped and checkerboard blocks

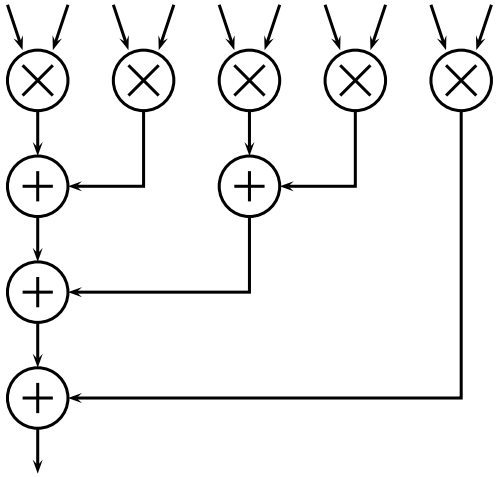

Figure 4.4. Reduction Tree with $N = 5$

adding $\frac{N}{2}$ resulting operand pairs in a second step and then consecutively adding resulting pairs until only one value results. This process is visualized in figure 4.4. The maximal increase of throughput is of factor $\log_2(N)$ and the latency can be decreased by factor $\frac{1}{\log_2(N)}$, assuming that N computation units are available and the memory interface is able to handle a throughput that is N times higher than in the sequential case.

The finalization step of the algorithm consists of calculating the likelihood. This is done by simply accumulating the N elements of the last forward variable α_L. This operation can be parallelized with a reduction tree, resulting in a throughput and latency optimization as described above.

The following sections will describe the impact on performance if scaling or the extension of the forward algorithm is implemented. Also the availability of parallelization in both cases will be discussed. Everything will be concluded with an overview of available parallelism and a discussion about the usefulness of each parallelization method in the context of the different algorithm implementations.

4.3 Prediction Model Simplification

Until now, it was always assumed, that the model is fully connected (ergodic), i.e. that every state can be reached from every other state. This is not necessarily the case, as it is often possi-

ble to describe a system with a simpler model. By adding more constraints to the possible state transitions (e.g. only one direction, feed-forward), only a few elements in the transition probability matrix are non-zero. In this case it is beneficial to use an array (adjacency list) instead of a matrix to represent the transition probabilities. Different methods have been proposed on how to store sparse matrices, but they are usually strongly dependant of the architecture and will hence be discussed only after the type of acceleration device has been chosen. A list with only non-zero elements instead of a sparse matrix reduces the necessary memory to store the data and makes a lot of computations superfluous. A Matrix-Vector multiplication parallelization as described in the previous section would not be beneficial anymore as a lot of computational units would be idle in most of the time.

4.4 Scaling and Data Representation

Scaling may be applied to prevent that the continuous multiplication of numbers smaller than one (e.g. probabilities) result in zero, because of the limited accuracy by digitally representing fractional numbers. Scaling does not influence the order of complexity of the algorithm. By introducing a scaling method as proposed in the reference work, the complexity of calculating one α_k vector goes from N^2 (no scaling) to $N^2 + 2N + 1$ (scaling), which is the same order $O(N^2)$. However, the introduction of scaling may increase the usage of recourses significantly: In order to scale α_k, the division operation is used to compute the scaling factor. Division is far more complex than multiplication and hence uses more recourses. Additionally, instead of the sequence likelihood (equation 3.8) the sequence log-likelihood (equation 3.11) needs to be computed, with the even more complex log operation.

In order to limit the amount of necessary division operations, it is beneficial to consider the following: Rather than scaling each element of α_k by dividing it by a scaling factor (N divisions), first the inverse of the scaling factor can be computed, which is then multiplied with each element of α_k (one division and N multiplications). Using N multiplication units, this operation can be parallelized.

To compute the log-likelihood, N log and N sum operations are necessary, in comparison to N sum operations for the likelihood. In terms of memory, the log-likelihood is more complex because the scaling coefficients of each α_k are used and need to be stored, while for the likelihood only the last set of forward variables α_L are used. The computation of the log-likelihood can be parallelized by using N units computing the log function and additionally by a reduction tree to speed up the accumulation.

Instead of using the proposed scaling method, a simpler scaling may be applied. By analyzing the operands, an average scaling factor can be computed. Using the knowledge, that all the operands are probabilities,

$$\sum_{i=1}^{N} \pi_i = 1, \sum_{j=1}^{N} tp_{ij} = 1, \sum_{j=1}^{M} b_{ij} = 1 \tag{4.2}$$

and doing the computation of the forward variables,

$$
\begin{aligned}
\hat{\alpha}_1 &= \hat{b} \cdot \hat{\pi} = \frac{1}{NM} \\
\hat{\alpha}_2 &= N \cdot \hat{\alpha}_1 \cdot \hat{tp} \cdot \hat{b} = N \cdot \frac{1}{NM} \cdot \frac{1}{N} \cdot \frac{1}{M} = \frac{1}{NM^2} \\
\hat{\alpha}_3 &= N \cdot \hat{\alpha}_2 \cdot \hat{tp} \cdot \hat{b} = N \cdot \frac{1}{NM^2} \cdot \frac{1}{N} \cdot \frac{1}{M} = \frac{1}{NM^3} \\
&\vdots \\
\hat{\alpha}_L &= \frac{1}{NM^L}
\end{aligned}
\tag{4.3}
$$

it can be computed, that assuming no precision loss at each computational step k, on average a scaling factor of $\frac{1}{M}$ is necessary in each step k. If the intermediate precision of the computational units is high enough to compensate for scaling to much or to few, this method is an easy solution to keep the values in an acceptable range. However, if the precision is not available (eg. if a fixed point data representation is chosen) a fixed scaling factor can cause an overflow (very bad because the result will be wrong) or an underflow (may be acceptable because it is only a loss of precision). In this case, rather than choosing an average scaling factor of $\frac{1}{M}$ it is safer to choose the scaling factor to be equal to the maximal possible scaling factor of all values of a specific event in B (scale $\max\big(B(O_k)\big)$). By doing this, the scaling factor will be to small and if L is big, the forward variables will still approach zero, only slower than without scaling. This is either acceptable because of a high precision, or another scaling factor must be computed to prevent this. The implemented solution will be explained in detail in chapter 5.

Another aspect to consider is the choice of data representation (floating point versus fixed point). This depends on one hand on the necessary precision and on the other hand on the choice of accelerator type. While general purpose hardware such as CPU, GPU and DSP (to some degree) offer an abstraction to make the representation type transparent to the developer, specialized hardware such as FPGA or ASCI offer no such abstraction. For the later devices, floating point operations increase the complexity of the hardware design and the necessary hardware resources considerably. In terms of performance, general purpose devices benefit also from a sparse usage of floating point values. The complexity of the software development however is only marginally or not affected at all by the choice of data representation.

If by choice, scaling is omitted, a fixed point representation will not be possible, due to the rapid convergence towards zero by continuously multiplying probabilities together. This implies, that by omitting scaling to save resources, a floating point representation must be used, which again increases the resource requirements or has a negative impact on performance (or both).

The trade-off between the choice of using scaling or not versus the choice of the precision and the data representation, will be analyzed in more detail in chapter 5, when the technology of the accelerator has been chosen.

4.5 Extension of the Forward Algorithm

The proposed extension uses a transition probability matrix which is not constant. For every arriving observation symbol, the matrix must be recomputed by using the delay of the symbol (equation 3.1) and the sum of different cumulative distribution functions (equations 4.5 and

3.5). To compute N^2 cumulative distribution functions is very expensive but it only needs to be computed once per d_k and can then be stored for later usage. A transition probability matrix can be used for the computation of L forward variables due to the continuous computation of likelihood values (as depicted in figure 4.1). This implies, that storage for L such matrices must be available or the matrix must be recomputed when needed and only the delay value is stored. In case of a pipelined architecture, the additional storage or computation is not necessary (cf. figure 4.2 and table 4.1). The computation of the transition probability matrix can be fully parallelized with $R*N^2$ computation units to perform a cumulative distribution function of d_k, where R is the number of different cumulative distribution functions necessary to model the behaviour of the events. The memory interface needs to be able to provide a throughput that is $R*N^2$ higher than in the sequential case (note that each cumulative distribution function takes several parameters as input. Eg. the normal cdf has the two parameters μ and σ). In previous chapters, R was always assumed to be equal to one in order to simplify the problem.

Note, that while the computation of one extended transition probability is independent of N or L, it is still a lot more complex than the simple multiply-accumulate operation necessary to calculate a forward variable (cf. the list below). Due to this, the computational units calculating the transition probability matrix must provide a lot more performance than the units necessary to compute the resulting forward variables in order to not limit the throughput. To get a rough impression of how expensive the computation of a distribution function is, the following list with three common examples is provided:

Exponential CDF

This distribution function describes the time between events in a Poisson process (events occur continuously and independently at a constant average rate). It is expressed as

$$F_{exp}(x) = \begin{cases} 1 - \exp(-\lambda x) & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{4.4}$$

Only the exponential function is needed, which is quite a complex function compared to a multiplication and could be problematic in a fully parallelized implementation (considering all parallelization options), but realizable in an implementation with less parallelism.

Lapalce CDF

This distribution is somewhat the extension of the exponential distribution and is also called double exponential distribution as it can be described as two exponential distributions put together (one flipped horizontally). It is expressed as

$$F_{laplace}(x) = \begin{cases} 1 - \frac{1}{2}\exp(-\frac{x-\mu}{b}) & \text{if } x \geq \mu \\ \frac{1}{2}\exp(\frac{x-\mu}{b}) & \text{if } x < \mu \end{cases} \tag{4.5}$$

In terms of complexity, for the Laplace distribution holds the same as for the exponential CDF.

Gaussian CDF

This is a very important distribution that is used in a lot of applications. It is used for real-valued random variables whose distributions are not known. The Gaussian (normal) cumulative distribution function cannot be expressed in terms of elementary functions, which is the reason why the special error function erf is used. It is defined by equation

3.7. Using integration by parts and the substitution $x = \frac{d_k - \mu}{\sigma}$, it can be expressed as

$$\Phi(x) = \frac{1}{2} + \frac{1}{\sqrt{2\pi}} \cdot \exp(-\frac{x^2}{2}) \cdot \left[x + \frac{x^3}{3} + \frac{x^5}{3 \cdot 5} + \cdots + \frac{x^{2n+1}}{3 \cdot 5 \cdots (2n+1)} \right] \tag{4.6}$$

This computation is very expensive. It comprises of an exponential function and an iterative approach (to achieve the necessary precision) including the power function and additions. If this distribution is chosen to describe the time between events, parallelization will be very challenging, in order to prevent this calculation from being the bottleneck.

Considering the huge computation power needed to fully parallelize the extension, it may be beneficial to use a very specialized unit (ASIC) just for the computation of the cumulative distribution function.

Independent of the distribution, to not reduce the throughput of the fully parallelized computation of the forward variables, a small pipeline of two stages must be built, where in the first stage the transition probability matrix is computed and in the second stage the forward variables.

The correction of the diagonal elements (cf. listing 4.3) can be maximal parallelized by using N reduction trees to compute the sum of rows and N subtracters to correct the diagonal elements.

4.6 Parallelization Options and Scalability

In the sections above, a lot of parallelization has been proposed. A maximal parallelization can hardly be achieved due to the immense requirement of recourses and is also not necessary because of dependencies. In a first step, a theoretical analysis (in terms of complexity order) of the different parallelization options and their combination is done. Then the results will be discussed and a choice will be made. Finally some conclusions about a finer degree of parallelization will be drawn in respect to the chosen architecture. Table 4.2 shows the pipelined architecture (cf. figure 4.2), the parallel architecture in the case of maximal row partitioning (cf. figure 4.3), the combination of both architectures and in the last column the combination of both architectures plus the reduction tree (cf figure 4.4). The basic and the extended forward algorithm are both of the same complexity order. As only the complexity order is considered, real computation times of different operations as well as scaling operations can be ignored. Due to simplification, it is assumed that $N = L$ and that by scaling the problem, N and L are both changing in the same order.

The two first columns show that the smaller latency of the parallel architecture comes at the price of a larger memory interface (simultaneous memory access is required). By combining both architectures (3rd column), the throughput can be increased by factor N at the cost of increasing the order of computation units. Adding also the reduction tree, throughput and latency are increased, resp. decreased by an additional factor of $\log N$. This comes again at the cost of increasing the order of computation units further. While the benefits are welcome, N^3 is simply to high to realistically implement such a solution. Already with a small N a huge server farm or thousands of FPGAs or GPUs would be necessary. Considering this, the reduction tree parallelization method will not be used as it gives the fewest benefits ($O(LogN)$) for its cost ($O(N)$). Computation units in an order of N^2 (2nd column) is feasible for small N by combining multiple devices. While this may be acceptable for a very important computation where lots of

Metric	Pipelined	Parallel	Both	Both & Tree
Computation Units	$O(N)$	$O(N)$	$O(N^2)$	$O(N^3)$
Memory Space	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
R/W Access	$O(1)$	$O(N)$	$O(N)$	$O(N)$
Throughput	$\times N$	$\times N$	$\times N^2$	$\times N^2 \log N$
Latency	$\times 1$	$\times \frac{1}{N}$	$\times \frac{1}{N}$	$\times \frac{1}{N \log N}$

Table 4.2. Comparison of architectures in terms of complexity for the forward algorithm

		Pipelined	Parallel	
Basic	Computation Units	L	N	
	Memory Space	$2N^2 + 2LN + N$	$2N^2 + 3N$	
	Read Access	4	$N + 2$	
	Write Access	1	N	
Extended	Computation Units	$L + C$	$N + C$	$(C + 1)N$
	Memory Space	$(P + 1)N^2 + 2LN + N$	$PL(1 + N^2) + 3N$	$(P + 1)N^2 + 3N$
	Read Access	5	$2N + 2$	
	Write Access	1	N	

Table 4.3. Pipelined versus parallel architecture for the basic and the extended forward algorithm (C: Number of computation units to compute one CDF, P: Number of parameters to compute one CDF)

people depend upon (eg. weather forecast, Google queries, etc.) failure prediction hardly falls into this category especially if failures of an embedded system are predicted. Additionally, using multiple devices in order to scale the problem, implies off-chip-communication and -memory. This will result in huge bottlenecks and have a huge impact on the actual speedup. This leaves the first two columns to compare for the application at hand (A combination of both methods can still be considered, but not by using maximal parallelization).

Table 4.3 now only compares the pipelined architecture with the parallel architecture but with more detailed estimations of resource usage. The comparison is done for the basic and the extended forward algorithm (for explanations refer to the previous sections in this chapter). In case of the extended forward algorithm, when the parallel architecture is used, there is a choice to be made if the parallelization should be achieved by increasing the memory usage or the number of computational units (hence the two columns).

The benefits of a parallel architecture over the pipelined are first and foremost the reduced latency and in case of the basic algorithm also the lower memory footprint. If the acceleration architecture of choice has a memory interface that allows the required throughput, for the basic algorithm this architecture should be chosen. In case of the extended algorithm this is only possible if enough computational units are available (and the CDF computation is not to complex) or if the on-chip memory is large enough to save transition probabilities for later use.

For the basic algorithm, the pipelined architecture should only be chosen if the memory interface becomes the bottleneck (for large N). Ideally a combination of the parallel architecture and the pipeline should be chosen in order to maximize the memory interface usage. By doing

this, a smaller latency is achieved by keeping the throughput high. Another reason to chose the pipelined architecture would be a simple state transition model (cf. section 4.3) which allows to save only the non-zero transition probabilities in a list. From this optimisation the pipelined architecture benefits on a much larger scale as less serial accumulations would be necessary, while in the parallel architecture only the utilization of the computational units would be reduced (less power consumption but no impact on performance).

In case of the extended forward algorithm, it is almost always better to choose the pipelined architecture: It uses either less memory or less computational units, allows optimization in case of simple models and allows more time to compute the transition probabilities. Parallelization is only possible for very simple CDF computations and for a small N.

The scalability is in both cases limited, but more so with the parallel architecture: If N or L becomes large such that off-chip memory is necessary (already for very small N or L for CPUs and GPUS, less so for more flexible architectures like FPGAs or ASICs) the memory interface will be to small to handle memory access simultaneously and hence become the bottleneck. The pipelined architecture does not have this drawback but has a slightly higher memory footprint. If memory can be handled on-chip but multiple chips are used to increase the number of recourses, both architectures can be scaled easily in L dimension (sequence length) but only with difficulty in N dimension (number of states) because the necessary communication links (all components of the vector α_{k+1} always depends on all components of the vector α_k). The scaling in dimension N of the pipelined architecture only depends on the memory usage, while the parallel architecture has a dependency of N for the computation units as well as for the memory.

4.7 Selection of an Appropriate Accelerator Type

better title?

For being able to choose an appropriate accelerator type, first a list with different acceleration options is presented. For each type the most common benefits and shortcomings are mentioned. At the end of the section a choice will be made using the following list and the observations discussed in the previous sections.

CPU

The Central Processing Unit falls into the class of the general purpose processors and is (usually) of the type SISD[1]. It is very flexible in terms of software interpretation (with the use of compilers) and allows to implement any kind of function in a fast, easy and maintainable fashion with no requirements in hardware knowledge. Operations like division, exponential function and logarithm are available as well as the floating point number representation. All operations can be executed with very high precision and at high clock frequencies (up to 3 GHz). CPUs come with the drawback of a high power consumption, limited parallelization options (small number of cores) and a fixed hardware architecture that causes big computation overheads (instruction pipeline, memory hierarchy, generalized computation units).

GPU

Like the CPU, the Graphics Processing Unit still falls into the class of general purpose

[1] Flynn's Taxonomy: Single Instruction, Single Data Stream

processors due to the hardware abstraction layers. A GPU is composed of a lot of small but quite powerful streaming processors of the type SIMD[2]. This processing power allows a lot of parallelization at a low price. As the CPU, also the GPU is very flexible in terms of software interpretation, requires however some hardware understanding in order to use the parallel power in an optimal way. Also a GPU allows to work with operations like division, exp and log functions and provides floating point number representation. GPUs can work with high precision and at high clock frequencies. The power consumption of a GPU is very high due to the high frequencies and the streaming processors. While the hardware abstraction layer provides flexibility in software and ease of use, it is the main reason for a computation overhead for basic operations. The fixed hardware architecture (especially the memory hierarchy) can prove to be a drawback in specific cases (eg. data usage provokes always cache misses).

DSP

The Digital Signal Processor is a specialized integrated circuit, largely used for digital signal processing. The key components of DSPs are optimized Multiply-Accumulate instructions, special SIMD operations and for DSP operation optimized memory architecture. They provide a hardware abstraction and are pretty easy to program. This causes some overhead as an instruction pipeline is necessary. The overhead is a lot smaller than in the case of CPUs because of specific instructions sets (this comes with a loss of flexibility). Fixed point as well as floating point devices exist with various precisions. A DSP provides a lot of specific computation power for a low price.

FPGA

The Field Programmable Gate Array is a customizable integrated circuit. It provides large recourses of logic gates as well as standard blocks such as RAM or highly optimized Multiply-Accumulate units. FPGAs provide high performance for very specific design optimized for one function. Once an FPGA is configured, as long as it keeps this configuration no other function will run on this device as it is the direct hardware representation of the function. This direct representation allows a very low overhead as operations are done directly in hardware without any instruction pipeline. FPGAs provide a high amount of hardware flexibility that is only toped by ASICs (see below). This flexibility comes at a medium price as FPGAs can be produced in big lots but are more difficult to produce than DSPs. A huge advantage is the possibility the build very big memory interfaces inside the chip and customize the memory architecture for the application at hand. The drawback of FPGAs is the increased development time necessary to implement a hardware solution of a function, the "low" clock frequency of up to 500MHz (this is low compared to CPUs or GPUs) and the absence of division, exponential and logarithm functions. All units are optimized for fixed point representation and floating point numbers must be implemented manually. Core generators and very powerful synthesis tools try to amend these drawbacks but still a very deep knowledge of hardware is necessary to successfully implement a design on FPGA.

ASIC

The Application Specific Integrated Circuit is, as the name already tells, an integrated circuit that has been designed for one (and only one) specific application. In comparison to the FPGA, an ASIC is built with fully customized elements and provides the full flexibility

[2]Flynn's Taxonomy: Single Instruction, Multiple Data Streams

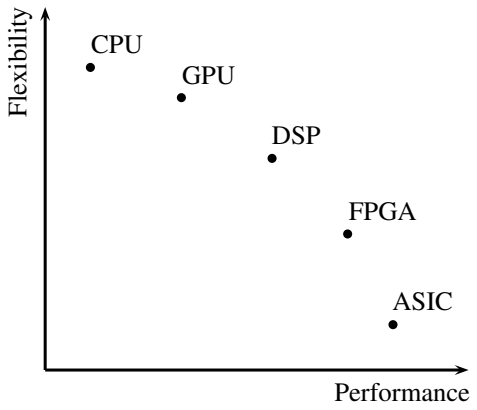

Figure 4.5. Flexibilty versus Performance of Hardware Devices

achievable with todays hardware knowledge. An ASIC has ideally no overhead as the hardware is a direct mapping of the function. This leads to very high performance at very low power consumption (as a rule of thumb a factor of 1000 can be assumed in either performance gain or power consumption decrease or a combination of both compared to CPUs). An ASICs is very expensive to produce. This includes the long development time and the production cost. Of course very deep hardware knowledge is necessary to create an ASIC of a specific function.

Comparing the accelerator types mentioned in the list above in a general manner, one can conclude that the further down the list one goes, the better is the performance - prower consumption ratio and the lower is the flexibility of the device to accept general function descriptions (cf figure 4.5, eg. a function description for a FPGA must be a lot more specific than one for a GPU but the performance-power consumption ratio is a lot better in case of the FPGA). For this work, one key point is high performance, because the algorithm to implement is computation intensive and needs to be executed fast. Another important point is the size of the system: the target application to predict failures is an embedded system, where space is usually limited. Power consumption may not be a main aspect, but it certainly needs to be considered to not exceed requirements of the main system.

Due to the limited parallelization options or the huge power and space requirements of CPUs, this accelerator type is not suitable for the application in question. GPUs provide a lot of computation power at a low price but have drawbacks in terms of architecture flexibility, space and power requirements. GPUs are a far better option than CPUs but may not be ideal because of the fixed memory interface. DSPs would be the device of choice if only the basic forward algorithm is considered. Due to the high performance at a low cost of DSPs, and the fact that the basic forward algorithm mainly uses multiply-accumulate operations one or multiple floating point DSP devices could be used to efficiently implement the algorithm. In case of the extended algorithm a DSP device causes to big an overhead to compute the CDF and will be the bottleneck of the design. For this reason a more flexible device must be chosen. The most flexible architecture is an ASIC. ASICs are very expensive in terms of money and development time, two recourses that are not available for this work. This leaves the FPGA: An FPGA combines the parallel power of DSPs for Multiply-Accumulate operations but adds the possibility to design a specific hardware architecture to compute the transition probabilities

needed for the extension. While the performance will not be at a level of an ASIC, the specialized hardware will still outperform any other device because of the customizable memory interface (even at the lower frequencies of FPGAs).

Chapter 5

Design and Implementation

Following the argumentation of the previous chapter, this chapter will describe the design of the pipelined architecture of the extended forward algorithm on a FPGA.

To design the accelerator, the top-down approach was applied: the algorithm is broken down into blocks, where each of them is broken down further until the basic functional blocs of the FPGA can be used for the implementation. The implementation follows then the bottom-up approach where each sub-block is implemented and tested. Completed blocks are grouped together to bigger blocks until finally there is only one big block remaining, describing the complete algorithm.

5.1 Architecture

The top architecture of the proposed algorithm is depicted in figure 5.1. The non-failure as well as all the failure sequence detection blocks denoted as "non-failure" and "failure type s" represent each the complete forward algorithm as it was described in the previous chapter. A system controller, denoted as "SYS CTRL" governs these blocks and the flash (FLASH CRTL)and memory controllers (RAM CTRL) load data from persistent respectively from volatile memory devices into the system. All the prediction blocks calculate a sequence likelihood and feed it to the classifier denoted as "classification". The classifier then decides if the present sequence leads to a failure or not and outputs this result as a boolean value. In terms of hardware, all the "failure" blocks are identical and can be run in parallel. They only differ by the values that are fed into the internal memory of the blocks.

In the following the designing details of the failure blocks are presented. The functionality of the pipelined architecture has already been described in chapter 4.2. The figure 4.2 depicts the basic schematics. The same architecture is shown in figure 5.2 with a simplified RTL (register transfer level) schematic. The main aspect of the chosen design is to use the high performance multiply-accumulate units available in modern FPGAs. These so called DSP-slices allow to perform, besides other operation, fully pipelined multiply-accumulate operations at frequencies of up to 450 MHz. The manual [30] describes the vast functional possibilities of the DSP-slices available in the series 7 FPGAs of Xilinx[1]. As shown in figure 5.2, in each pipeline stage the transition probabilities (coming either from memory in case of the basic forward al-

[1] http://www.xilinx.com

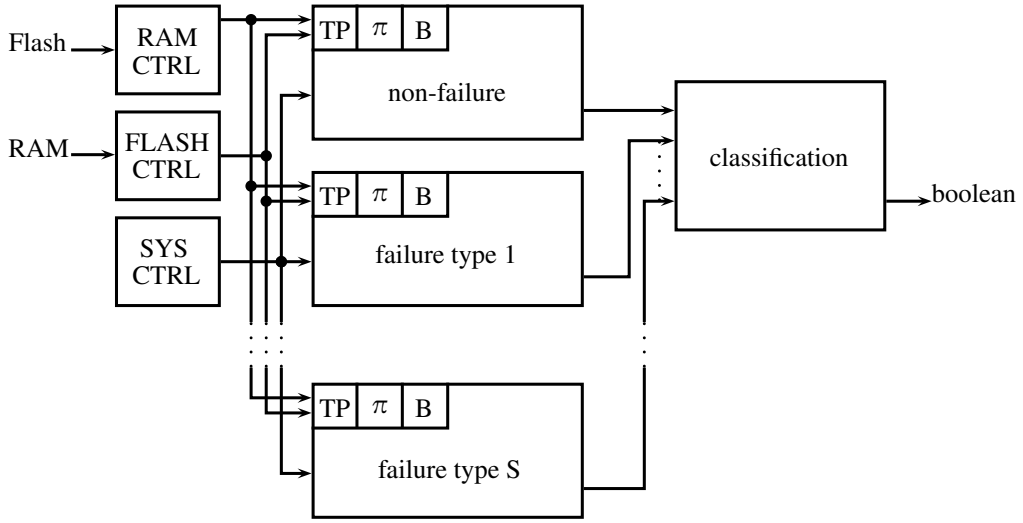

Figure 5.1. Top architecture of the failure prediction algorithm

gorithm or from a computational unit in the case of the extended algorithm) are multiplied with the forward variables (read out of the fifo queue) calculated in the previous stage and the results are accumulated. This is done until N components are accumulated, then the emission probabilities are multiplied to the result and the first component of the forward variable vector α_k of this stage is stored into a fifo queue of the next pipeline stage. This iteration is repeated N times until all N component of the α vector are computed. After each iteration, the accumulator must be cleared. It takes three pipeline stages to pipeline the multiply- accumulate operation and two stages for the multiply operation. It takes $N+3$ cycles to accumulate all multiplication pairs ($\alpha_{k-1,j} * tp_{i,j}$) and 2 cycles to perform the final multiplication. This adds up to N^2+5) cycles to compute all components of the vector α_k (the setup time needs only to be considered once if a well timed reset of the accumulator is performed).

With this design, there are several things to note. A first concern focuses on the second multiplier of the chain: It is used only once every $N+3$rd cycle and computes either nonsense or is idle (if disabled) during the rest of the time. This is a very poor utilization of recourses and can be optimized by reusing the first multiplier to do the second multiplication. Doing this increases the necessary cycles to a minimum of $N*(N+3)+2$ cycles but cuts the necessary multiplication units (DSP-slices) by half. Another point to note concerns the FIFO queue. The queue is supposed to store the arriving components of the vector α_{k-1} and use them to compute the next vector α_k. However, each component of the vector α_{k-1} is used N times, as every component of the new vector α_k depends on every component of the previous vector α_{k-1}. Therefore it is not possible to use only one FIFO queue. A solution to this problem is to use two queues, one to store the arriving new values needed for the next iteration and one to read the values that have been stored last iteration. At the end of the computation of all components of the vector α, the queues are switched. While a queue is in read state, it operates as a circular buffer by storing back the value it has just read. Another solution is to use addressable memory blocks instead of FIFOs and use a simple counter to increasingly address the required value to read or to store a new vale. While the solution with the FIFOs is easier to control, the memory block solution offers another big advantage: if a simple model for the state transitions is used

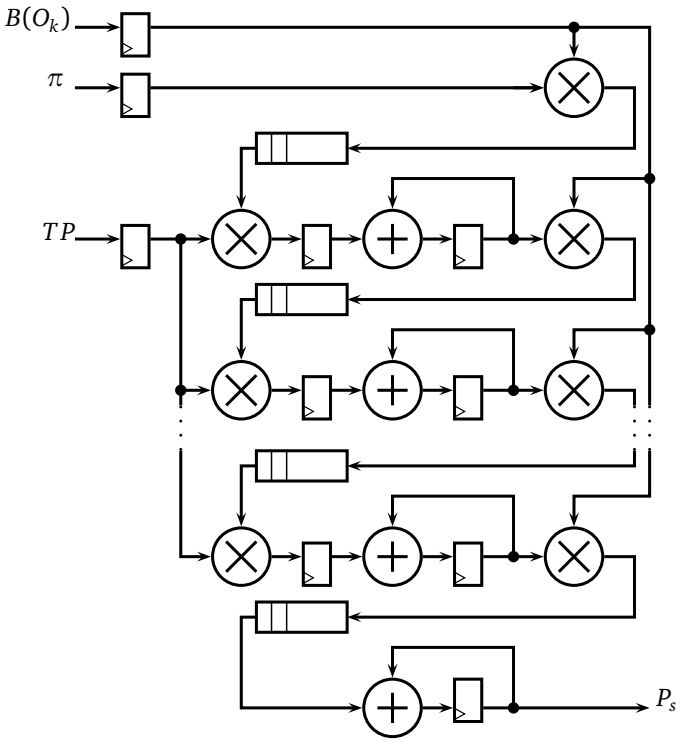

Figure 5.2. Simplified RTL representation of the pipelined forward algorithm

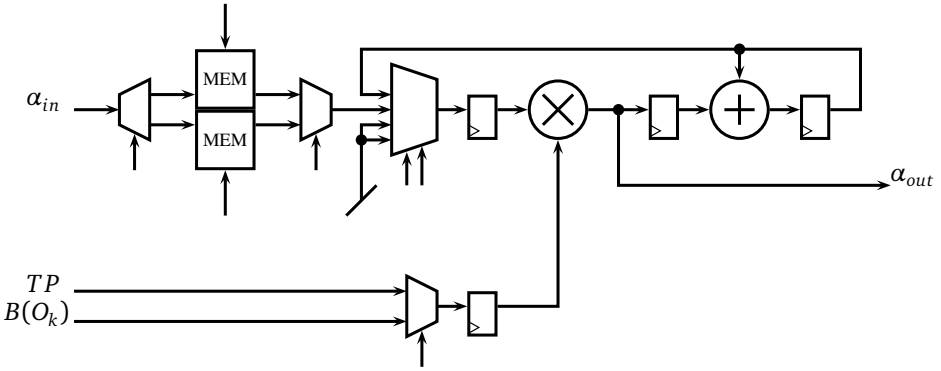

Figure 5.3. RTL implementation of a pipeline stage with dual memory queue and reusage of components

(as discussed in chapter 4.3), a lot of components of the transition probability matrix will be zero. It doesn't make sense to compute the product of zero with a component of the α vector as the result will always be zero and won't impact the accumulation. Using an addressable memory block, the α vector components corresponding to a zero in the transition probability matrix could easily be skipped while with the FIFOs, this would not be possible. Choosing an appropriate method to store the sparse matrix in memory, the computation of the new address to read the next α vector component corresponding to a non zero transition probability can be done very fast and with only few recourses (see chapter 5.3 for more information).

Figure 5.3 shows the detailed design of one pipeline stage, if the multiplication unit (DSP-slice) is reused and if to memory blocks are used, alternating one to write new data and one to read data stored from the last α vector computation. The control signal are not labeled here for reasons of readability and will be explained in detail in chapter 5.4.

describe the design of the extension

5.2 Operand Scaling and Precision

The DSP-Slices used to for multiply and accumulate operations are designed for fixed point data representation, hence thy do not support floating point operations. This poses no problem if an operand width can be chosen that allows to represent the complete spectrum of all possible values an operand can take (with respect to a certain precision). The continuous multiplication of probabilities, as it is the case with the algorithm at hand, lead to very small values very fast, depending on the sequence length L. This is the reason why a fixed point representation without scaling cannot be used. This can be circumvented by using floating point numbers with a very large exponent to prevent an underflow. In this case, the DSP-Slices need to be extended in order to support floating points. In the following this option is discussed and then compared to a fixed point solution with scaling.

In order to multiply two floating point numbers, the mantissas of both numbers are multiplied and the exponents are added (the sign can be ignored, as probabilities are always positive). For the multiplication of the mantissas the DSP-Slices can be used. In parallel to this operation, an external adder can add up the exponents. For being able to add floating point numbers (which should be done with the accumulator of the DSP-Slice) the exponents of the

numbers must be equal. To achieve that, the difference of the two exponents is calculated and then the mantissa of the number with the lower exponent must be shifted by this difference (and possibly rounded/truncated). This process is called normalizing. After the normalization, the mantissas can be added and finally the resulting value needs to be normalized again.

The work [3] presents a solution for a floating point operations, including multiplication and addition, in single precision[2] using only one DSP-Slice. For being able to omit the scaling completely, the exponent would need to be larger: assuming that the alphabet contains $M = 1000$ elements and using the estimation of equation 4.3, a sequence length of $L > 13$ would already exceed the capabilities of single precision data representation. The drawback of this solution is the dependency of the exponent on the sequence length L and the non-standardized representation. The proposed solution would needed to be extended by a parametrizable exponent size. Another issue in this work is the absence of a optimized and fully pipelined multiply-accumulate operation. In order to do multiplication followed by accumulation with the same DSP-Slice, one operation must first finish completely before the other can begin. This has a huge impact on the latency as well as on throughput: Using the latencies presented in the work, 22 cycles are necessary for multiplication and 25 cycles for addition. Using memory to store the intermediate results the operations could be pipelined, but without additional memory no pipelining is possible. A more optimized solution for the algorithm at hand can surely be found but a more profound analysis of floating point operations on FPGA must be performed to gain more insight. This is however beyond of the scope of this work and the present facts are enough to first consider a fixed point solution with scaling before heading deeper into floating points.

The documents [27, 10] discuss the choice of fixed point versus floating point representation with respect to DSP devices. This is also applicable for FPGAs.

use the argumentation of those documents

The DSP-Slices allow fixed point multiply-accumulate operations with a operands of bit width 18 and one of bit width 25. Internally the device works with a bit width of 48 bits, which allows a lossless multiplication (43 bits) and an accumulation with 5 bits margin for overflow. Due to the properties expressed in equations 4.2, these fife additional bits are not necessary in the present case. After the MACC operation, the second multiplication takes a gain operands of with 18 bits resp 25 bits as input. Therefore, the result of the MACC needs to be truncated. In order to not loose to much information, at this step scaling must be introduced.

As already discussed in chapter 4.4 the scaling method proposed by the reference work must be avoided in this implementation due to the additional necessary operations. Instead, a scaling in base of 2 can be used (this corresponds to a shift): after the computation of a component of the initial α vector, a lead zero counter (LZC) unit is introduced. This unit is purely combinational and counts the leading zeros of a operand. The minimal count of all components in the α vector is then forwarded into the next pipeline stage where it is used to to shift the result of the MACC operation for LZC positions to the left (this corresponds to a multiplication of 2^{LZC}). Also in this stage (and every following) a LZC unit is added in the same way as described above. The LZC results of each stage are forwarded and used to shift the results and accumulated. The minimum LZC of all elements of the α vector has to be found because all operands of one stage need to be scaled by the same value. It must be the minimum value in order to prevent a overflow in the shifting operation. It is important to use the same value for all operants to being able to accumulate the operands without normalization in the next pipeline stage.

[2] IEEE 754-2008

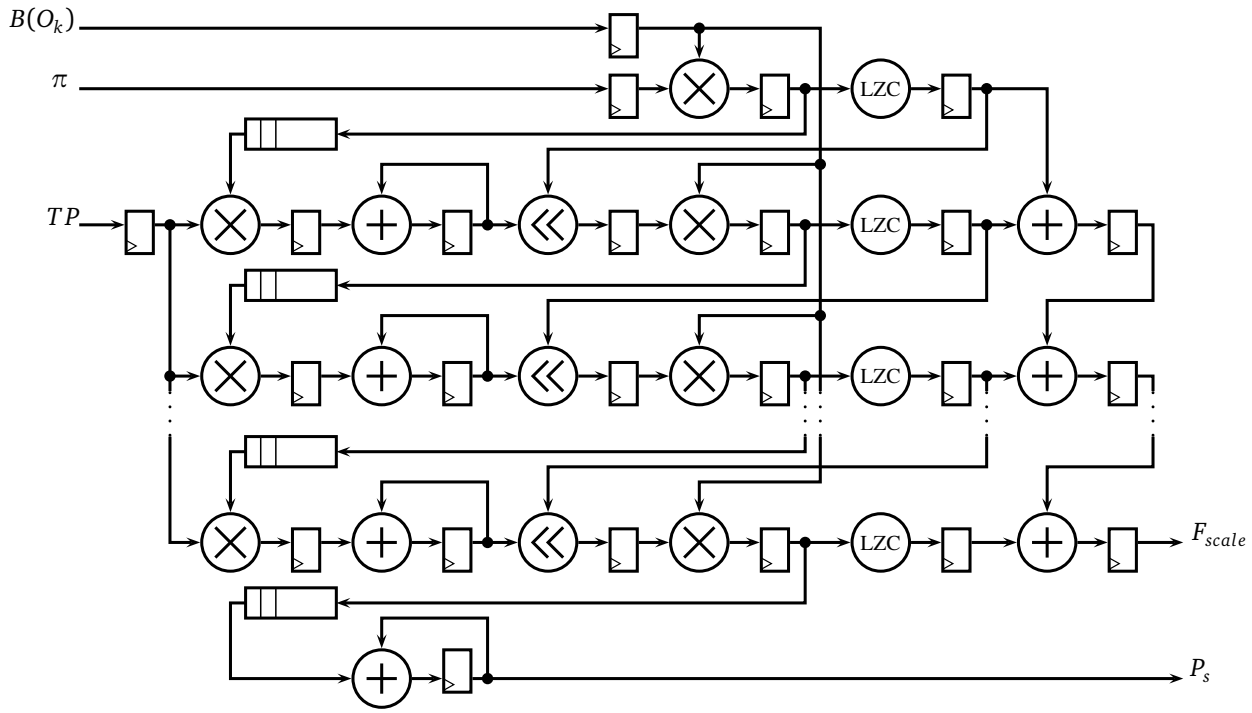

Figure 5.4. Simplified RTL representation of the pipelined forward algorithm with scaling using shifters and leading zero counters

Simultaneously to the resulting likelihood, also a scaling factor is provided. This scaling factor can then be used in the classification step to normalize all the involved likelihood values and decide if a failure is predicted for the current event sequence. The scaling method is presented in figure 5.4.

To perform the scaling, also a DSP-Slice can be used by simply multiplying the operand with the factor 2^{LZC}. The whole range of internal 43 bits of the MACC operation need to be scaled which can be achieved by two cycles of one DSP-Slice. First the lower 18 bits (op_{low}) of the 43 internal bits (op) are introduced into the DSP-Slice and multiplied by the factor 2^{LZC}. A precomputed lookup table is used to select the factor corresponding to 2^{LZC} using LZC as input. In next cycle, the upper 25 bits (op_{up}) of op are selected and introduced into the DSP-Slice. op_{up} is shifted by a constant amount of 17 bits (this can be done internally by the DSP-Slice) and then also multiplied by the factor 2^{LZC}. The accumulator then adds the two shifted operands op_{low} and op_{up} together which results in a shifted value of op by LZC bits to the right. All these operation can be done by using only one DSP-Slice. Figure 5.5 shows the necessary schematics. Also the simplified representation of the memory queue has been replaced in this schematic.

To select the minimal LZC value, two registers and a comparator are used. The first register stores the new LZC value and the second register holds the output LZC value. The output register is only updated with the new value if the new value is smaller than the one already stored in the output register.

To further increase the precision, the stored values π, B and TP can be preprocessed and scaled. Also here, the element with the lowest number of leading zeros decides the scaling

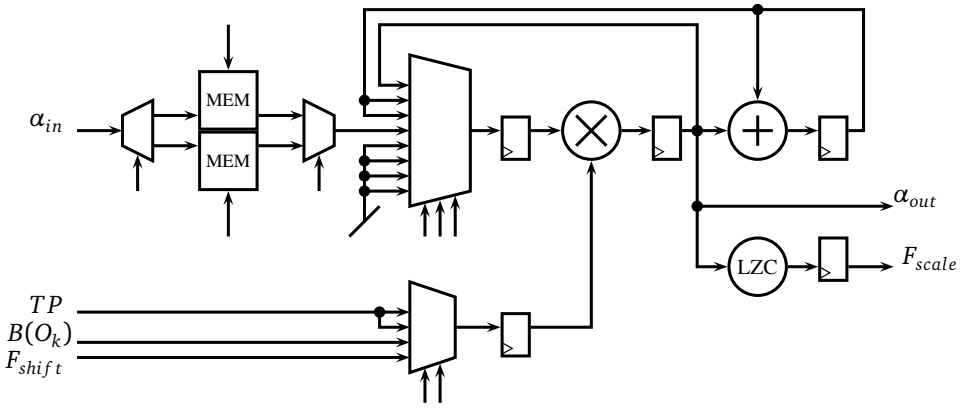

Figure 5.5. RTL implementation of a pipeline stage including scaling

factor.

calculation of quantization error, conclusion that fixed points are fine

5.3 Memory Architecture and Management

internal block ram of the FPGA for π, TP and B. In case of the extension, instead of TP the cdf parameters are stored.

π and TP/cdf parameter values can be stored once and for all (they never change)

B changes depending on the arriving event. There are usually to much different events ($M = 1000$) to store all inside the FPGA. A two staged pipeline is necessary to keep the throughput at a maximum level. In case of the extension this top level pipeline already exist. In the first stage of the pipeline, a memory controller reads from an external memory the B values corresponding to the actual event and stores them into the internal buffer. This buffer is of the type FIFO. This needs to be done for every active prediction model (cf. figure 5.1)

The events do not arrive in a regular interval. They may arrive very fast one after each other and there my be time intervals where no event arrives. No hard real time constraints for the prediction system exist: There is no impact on the prediction quality, if the latency from the time an event arrives until the likelihood is computed differs depending on the load of the system. This can be used to even out the event stream by introducing a FIFO queue where the events are stored until the system is able to compute the corresponding likelihood. This buffer needs to be designed in order to be able to collect events when the accelerator is used to full capacity and events are arriving faster than the accelerator can handle. Ideally, the buffer should never be empty and never full. A buffer that is always empty, indicates that the performance of the accelerator is is to high. The frequency of the accelerator can be lowered in order to save more energy. If the buffer gets filled up, it becomes problematic, because new events cannot be stored and are lost. If the average frequency of the arriving events is lower than the maximal throughput the buffer is designed to small. If the frequency is higher, the performance of the accelerator is insufficient.

As already mentioned in chapter 5.1, intermediate vectors α_k need to be stored in memory, as each value is used multiple times to compute the next vector α_{k+1}. FPGAs provide internal memory blocks that can be accessed in one clock cycle. These memory blocks limit the maximal

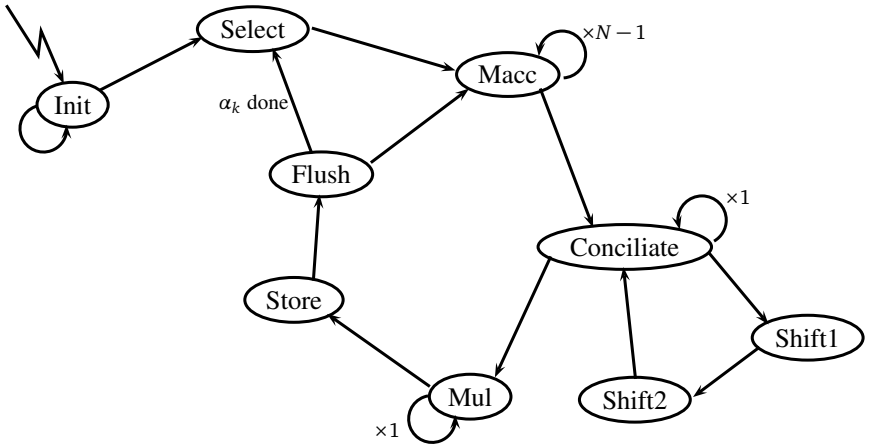

Figure 5.6. Mealey State Machine to control the pipeline

possible values of N and L to be implemented on one FPGA. L is also limited by the available MACC units. Table 4.3 shows the necessary memory in function of N and L.

how to store sparse matrices (wrt chapter 5.1)

5.4 Controlling the Pipeline

*in table **??** the signals flush_ps and load_out are only set in the first iteration. add a footnote?*

5.5 Balancing the Pipeline Stages

5.6 Implementation and Testing

- generic design (change n, l and bit widths in param_pkg)
- if bit widths of op1 is changed, fifo_512x25 must be regenerated

type	signal	init	select	macc	conciliate	shift1	shift2	mul	store	flush	input
	pi_we										input
	tp_we										input
	b_we										input
	data_ready										input
1	enable_count	0	0	0	0	0	0	0	1	0	pi_we
2	enable_ctrl	0	0	0	0	0	0	0	0	0	data_ready
3	enable_init	0	0	0	0	0	0	1	0	0	data_ready
3	enable_init_mul	0	0	0	0	0	0	1	0	0	data_ready
4	enable_step	0	0	1	1	1	1	1	0	0	data_ready
4	enable_step_macc	0	0	1	1	1	1	1	0	0	data_ready
5	enable_final	0	1	0	0	0	0	0	0	0	data_ready
4	laod_op2	0	0	1	1	1	1	1	0	0	data_ready
6	load_step_alpha	0	0	1	0	0	0	0	0	0	
7	load_final_alpha	0	0	0	0	0	0	0	1	0	
8	load_scale_new	0	1(d)	0	0	0	0	0	0	0	
9	load_scale_acc	0	1	0	0	o	0	0	0	0	
7	store_init_scale_new	0	0	0	0	0	0	0	1	0	
23	store_init_scale_small		1(d)								or internal
9	store_init_scale_ok	0	1	0	0	0	0	0	0	0	
7	store_step_alpha	0	0	0	0	0	0	0	1	0	
7	store_step_scale_new	0	0	0	0	0	0	0	1	0	
23	store_step_scale_small		1(d)								or internal
9	store_step_scale_ok	0	1	0	0	0	0	0	0	0	
5	store_final_ps	0	1	0	0	0	0	0	0	0	data_ready
10	store_final_ps_delayed	0	1(d)	0	0	0	0	0	0	0	data_ready
10	store_final_scale	0	1(d)	0	0	0	0	0	0	0	data_ready
11	shift_step_acc	0	0	0	1(6)	0	0	0	0	0	
12	sel_mux2_op2	0	0	0	0	0	0	1	0	0	
13	sel_step_read_fifo		(*)								special
12	sel_step_op1	0	0	0	0	0	0	1	0	0	
14		0	0	0	1	1	0	0	0	0	
15		0	0	0	1	0	1	0	0	0	
16	sel_step_op2	0	0	0	1(5)	1	1	0	0	0	
17	flush_init	0	0	0	0	0	0	0	0	1	
17	flush_step_macc	0	0	0	0	0	0	0	0	1	
18	flush_step_acc	0	0	0	0	0	1	0	0	0	
9	flush_step_fifo	0	1	0	0	0	0	0	0	0	
	reset_n	0	0	0	0	0	0	0	0	0	input
19	reset_count_n	0	0	0	0	0	0	0	0	0	reset_n
19	reset_op2_n	0	0	0	0	0	0	0	0	0	reset_n
19	reset_ctrl_n	0	0	0	0	0	0	0	0	0	reset_n
19	reset_init_n	0	0	0	0	0	0	0	0	0	reset_n
20	reset_init_mul_n	0	0	0	0	0	0	0	0	1(n)	reset_n
19	reset_step_n	0	0	0	0	0	0	0	0	0	reset_n
20	reset_step_macc_n	0	0	0	0	0	0	0	0	1(n)	reset_n
21	reset_step_fifo0	0	1(*(n))	0	0	0	0	0	0	0	reset
22	reset_step_fifo1	0	1(*)	0	0	0	0	0	0	0	reset
19	reset_step_scale_new_n	0	0	0	0	0	0	0	0	0	reset_n
19	reset_step_scale_small_n	0	0	0	0	0	0	0	0	0	reset_n
19	reset_step_scale_ok_n	0	0	0	0	0	0	0	0	0	reset_n
19	reset_init_scale_new_n	0	0	0	0	0	0	0	0	0	reset_n
19	reset_init_scale_small_n	0	0	0	0	0	0	0	0	0	reset_n
19	reset_init_scale_ok_n	0	0	0	0	0	0	0	0	0	reset_n
19	reset_final_n	0	0	0	0	0	0	0	0	0	reset_n

Table 5.1. detailed control signals

Chapter 6

Results

- nexys4 board with artix-7 fpga
- limited recourses -> proof of concept
- bord hardware for testing

6.1 Speedup

6.2 Accuracy

Chapter 7

Conclusion

7.1 Main Contribution

7.2 Future Work

Appendix A

some material

Bibliography

[1] D. ANGUITA, A. BONI, AND S. RIDELLA, *A digital architecture for support vector machines: theory, algorithm, and FPGA implementation*, IEEE Transactions on Neural Networks, 14 (2003), pp. 993–1009.

[2] M. AZHAR, M. SJALANDER, H. ALI, A. VIJAYASHEKAR, T. HOANG, K. ANSARI, AND P. LARSSON-EDEFORS, *Viterbi accelerator for embedded processor datapaths*, in IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP, July 2012, pp. 133–140.

[3] F. BROSSER, H. Y. CHEAH, AND S. FAHMY, *Iterative floating point computation using FPGA DSP blocks*, in International Conference on Field Programmable Logic and Applications, FPL, Sept 2013, pp. 1–6.

[4] S. CADAMBI, I. DURDANOVIC, V. JAKKULA, M. SANKARADASS, E. COSATTO, S. CHAKRADHAR, AND H. GRAF, *A massively parallel FPGA-based coprocessor for support vector machines*, in IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM, April 2009, pp. 115–122.

[5] S. CHE, J. LI, J. SHEAFFER, K. SKADRON, AND J. LACH, *Accelerating compute-intensive applications with GPUs and FPGAs*, in Symposium on Application Specific Processors, SASP, June 2008, pp. 101–107.

[6] D. CLARKE, A. LASTOVETSKY, AND V. RYCHKOV, *Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models*, in Euro-Par 2011: Parallel Processing Workshops, M. Alexander, P. D'Ambra, A. Belloum, G. Bosilca, M. Cannataro, M. Danelutto, B. Di Martino, M. Gerndt, E. Jeannot, R. Namyst, J. Roman, S. Scott, J. Traff, G. VallÃľe, and J. Weidendorfer, eds., vol. 7155 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 450–459.

[7] A. DEFLUMERE, A. LASTOVETSKY, AND B. BECKER, *Partitioning for parallel matrix-matrix multiplication with heterogeneous processors: The optimal solution*, in IEEE International Parallel and Distributed Processing Symposium Workshops PhD Forum, IPDPSW, May 2012, pp. 125–139.

[8] J. DETREY AND F. DE DINECHIN, *A parameterized floating-point exponential function for FPGAs*, in IEEE International Conference on Field-Programmable Technology, ICFPT, Dec 2005, pp. 27–34.

[9] C. DOMENICONI, C.-S. PERNG, R. VILALTA, AND S. MA, *A classification approach for prediction of target events in temporal sequences*, in Principles of Data Mining and Knowledge

Discovery, T. Elomaa, H. Mannila, and H. Toivonen, eds., vol. 2431 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2002, pp. 125–137.

[10] F. GENE AND S. RAY, *Comparing fixed- and floating-point dsps*, tech. rep., Texas Instruments Incorporated, Post Office Box 655303 Dallas, Texas 75265, 2004.

[11] A. JACOB, J. LANCASTER, J. BUHLER, AND R. CHAMBERLAIN, *Preliminary results in accelerating profile hmm search on FPGAs*, in IEEE International Parallel and Distributed Processing Symposium, IPDPS, March 2007, pp. 1–8.

[12] D. H. JONES, A. POWELL, C.-S. BOUGANIS, AND P. Y. K. CHEUNG, *GPU versus FPGA for high productivity computing*, in International Conference on Field Programmable Logic and Applications, FPL, Washington, DC, USA, 2010, IEEE Computer Society, pp. 119–124.

[13] E. KADRIC, P. GURNIAK, AND A. DEHON, *Accurate parallel floating-point accumulation*, in IEEE Symposium on Computer Arithmetic (ARITH), ARITH, April 2013, pp. 153–162.

[14] S. KESTUR, J. DAVIS, AND E. CHUNG, *Towards a universal FPGA matrix-vector multiplication architecture*, in IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM, April 2012, pp. 9–16.

[15] S. KESTUR, J. D. DAVIS, AND O. WILLIAMS, *Blas comparison on FPGA, CPU and GPU*, in IEEE Symposium on VLSI, ISVLSI, Washington, DC, USA, 2010, IEEE Computer Society, pp. 288–293.

[16] T.-T. LIN AND D. SIEWIOREK, *Error log analysis: statistical modeling and heuristic trend analysis*, IEEE Transactions on Reliability, 39 (1990), pp. 419–432.

[17] T.-T. Y. LIN, *Design and evaluation of an on-line predictive diagnostic system*, PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[18] C. LIU, *cuhmm: a cuda implementation of hidden markov model training and classification*, tech. rep., Johns Hopkins University, Mai 2009. Project Report for the Course Parallel Programming.

[19] R. P. MADDIMSETTY, J. BUHLER, R. D. CHAMBERLAIN, M. A. FRANKLIN, AND B. HARRIS, *Accelerator design for protein sequence hmm search*, in International Conference on Supercomputing, ICS, New York, NY, USA, 2006, ACM, pp. 288–296.

[20] E. NEUMANN, *Berechnung von hidden markov modellen auf grafikprozessoren unter ausnutzung der speicherhierarchie*, diploma thesis, Humboldt University of Berlin, Berlin, Germany, Mai 2011.

[21] A. OLINER, A. KULKARNI, AND A. AIKEN, *Using correlated surprise to infer shared influence*, in IEEE/IFIP International Conference on Dependable Systems and Networks, DSN, June 2010, pp. 191–200.

[22] T. OLIVER, L. YEOW, AND B. SCHMIDT, *High performance database searching with HMMer on FPGAs*, in IEEE International Parallel and Distributed Processing Symposium, IPDPS, March 2007, pp. 1–7.

[23] R. POTTATHUPARAMBIL AND R. SASS, *Implementation of a cordic-based double-precision exponential core on an FPGA*, Proceedings of RSSI, (2008).

[24] F. SALFNER, *Event-based Failure Prediction*, PhD thesis, Humboldt-University of Berlin, February 2008.

[25] F. SALFNER, M. LENK, AND M. MALEK, *A survey of online failure prediction methods*, ACM Comput. Surv., 42 (2010), pp. 10:1–10:42.

[26] F. SALFNER AND P. TRÖGER, *Predicting Cloud Failures Based on Anomaly Signal Spreading*, in Dependable Systems and Networks, IEEE, 2012.

[27] S. W. SMITH, *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing, San Diego, CA, USA, 1997, ch. Digital Signal Processors, pp. 514–520.

[28] R. VILALTA AND S. MA, *Predicting rare events in temporal domains*, in IEEE International Conference on Data Mining, ICDM, December 2002, pp. 474–481.

[29] J. WALTERS, V. BALU, S. KOMPALLI, AND V. CHAUDHARY, *Evaluating the use of GPUs in liver image segmentation and hmmer database searches*, in IEEE International Parallel and Distributed Processing Symposium, IPDPS, May 2009, pp. 1–12.

[30] XILINX, *7 Series DSP48E1 Slice*, 1.6 ed., August 2013.

[31] H. YANG, S. ZIAVRAS, AND J. HU, *FPGA-based vector processing for matrix operations*, in International Conference on Information Technology, ITNG, April 2007, pp. 989–994.

[32] C. YILMAZ AND A. PORTER, *Combining hardware and software instrumentation to classify program executions*, in ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE, New York, NY, USA, 2010, ACM, pp. 67–76.