Event-based Failure Prediction Accelerator

Acceleration of an Extended Forward Algorithm for Failure Prediction

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. Dr. Miroslaw Malek

July 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, 7. July 2014

Abstract

A large part of the society depends on technology. It is important that such technology is dependable. In order to make systems dependable, various techniques are available. One such technique is failure prediction. While failure prediction alone will not increase the dependability, it provides valuable information to facilitate the increase of availability and reliability of systems. The anticipation of upcoming failures allows to perform countermeasures in order to prevent systems from reaching a failure state.

This thesis proposes an accelerator design on a Field Programmable Gate Array (FPGA) for a failure prediction algorithm. The prediction method is based on the processing of error event sequences and modelled with a Hidden Semi-Markov Model (HSMM). The sequence processing is based on the forward algorithm. The design and implementation of the forward algorithm is provided in a fixed-point data representation and proposes a simple operand scaling method with a low overhead. The design focuses on low resource utilisation and energy consumption. The accelerator outperforms conventional processors by a factor of up to 48.85.

Acknowledgements

First, I want to thank Prof. Dr. Miroslaw Malek for giving me the possibility to do the thesis and providing me with constructive feedback and helpful suggestions.

I thank Samuel Schürch and Nicolas Lenz for reading the thesis, providing helpful advice and pointing out flaws.

Further I want to thank my girlfriend Giovina Nicolai, my flatmates, and my family who endured my unsteady mood during the final phase of the thesis.

Finally I thank all the brilliant novel writers out there I thoroughly missed during the last couple of months.

Contents

Figures

Tables

Chapter 1

Introduction

Nowadays it is increasingly important that computer systems are dependable. Computer systems are used more and more in areas where failure can lead to catastrophic events. Banking, public transportation, and medical engineering are only a few examples of areas where large and complex systems are employed. 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 (SW) comes without any faults. On top of that, the same problem arises with the Hardware (HW) components which may contain faulty parts and get increasingly prone to failures due to decay of material.

In the event of a system failure, it is 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. checkpointing (create checkpoints to roll back/forward), failover (switch to a redundant system), reboot. All these techniques require a certain amount of time to complete the recovery process. This time is very valuable because during the recovery of the system, it cannot provide the required service. In order to minimize this time, techniques have been developed to anticipate upcoming failures. The survey [25] categorizes and describes several proposed failure prediction techniques. One specific technique will be the reference work for this thesis and is described in [24].

The accuracy of the proposed algorithm to predict failures proves to be better compared to the techniques [17, 29, 9]. However, it has the drawback of increased complexity and hence increased computation time. It is very important to keep the computation overhead 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 HW accelerator for the prediction algorithm. The design of such an accelerator is outlined in this thesis.

1.1 Problem Statement

The main idea of the prediction model proposed in the reference work [24] is to predict failures based on sequences of events with their time of occurrence. This is modelled with a Hidden Semi-Markov Model (HSMM). The fundamental concept of this model is described in Chapter 3. A detailed description can be found in [24]. Three steps are required to perform a prediction: the training of the model, the sequence processing, and the classification. The training of the model uses collected error event data samples and corresponding oracle predictions, collected

from known failures, to train the parameters of the model. The training of the model is not time critical and is performed off-line (not on a live system). The trained model is then able to detect sequences, similar to previous sequences, that led to failures before. This is the second step, the sequence processing. This step is time critical as it happens on-line on a live system. The system will continuously send error events to the sequence processing unit which needs to compute a likelihood of the sequence of the last L events. This likelihood describes the probability of the observed sequence to be similar to one that led to a failure in previous cases. The likelihood is sent to the last step, the classifier, where it is compared to a non-failure sequence likelihood. The classifier then predicts if the sequence at hand will lead to a failure in the near future.

The failure prediction method, as it was outlined in very few words above and discussed in [24], leads to very good prediction results (cf. Chapter 3 for more information). This comes at the price of high computational efforts that are necessary to perform the sequence processing. This may lead to situations where a failure is correctly predicted, but the result of the prediction is only available after the failure has already occurred. This would rend the prediction useless. The main goal of this thesis, is to accelerate the sequence processing step in order to predict a failure before it happens. The acceleration will be achieved by running the prediction algorithm on specialized HW. A common acceleration method is to split the algorithm in multiple parts and run each part, in parallel, on separate computational units. To do so, first available parallelism and an appropriate accelerator architecture must be found. Then, based on the analysis, the accelerator must be designed and implemented. Finally the acceleration design must be compared to a serial implementation, and the speedup and accuracy must be computed.

1.2 Motivation

As outlined before, it becomes increasingly important to provide dependable infrastructure for information systems. Failure prediction is a very hot research topic with a main focus on server infrastructure. The proposed prediction algorithm targets single server nodes. With the recent trend of moving everything to the cloud, one may question the usefulness of failure prediction of single nodes. Rather than focusing on single machines it may prove beneficial to only consider the interaction of nodes and predict failures on a bigger scale. Why then designing an accelerator for a prediction model that targets single server systems?

Despite the trend of moving to the cloud, there are still thousands of single node systems that have a huge impact on a vast amount of people where failures would prove to be catastrophic. The talk is about embedded systems. Embedded systems are everywhere and the number is increasing. While server farms are growing, there is also the trend of specializing information systems. With the advances of computer HW, performance is available with less space and energy requirements, and hence machines (e.g. industrial devices, transportation, surveillance) are getting more intelligent with the increasing trend to embed powerful systems. Transportation (trains, planes, cars) is a prime example of dependable embedded systems. Huge efforts have been made to make fully autonomous transportation systems failure prone (e.g. bullet train in Japan) by adding redundant resources. Accurate failure prediction systems may be a key to reduce these resources or to increase the dependability.

Another issue with the proposed failure prediction method is the high amount of parameters that need to be identified, estimated, and set. Even if this initial work is done, the process must be repeated after some time interval, due to the fact that properties of server system change during their lifetime. This is usually not the case for embedded systems. Once an embedded

system is designed and installed, SW changes are a lot less common (neglecting the consumer market which is not the main target here), hence the system parameters do not change often or not at all. It is therefore sufficient to do the parameter configuration only once during design time. Another aspect is the specific application domain of embedded systems. They are designed for very specific functions and offer a better insight in process properties than this is the case with complex SW architectures on server systems. A parametrization of the failure prediction system for an embedded system will in general prove to be easier than for server systems.

These are the reasons why this work presents an acceleration for the failure prediction method proposed by [24]. The accelerator will be designed with special attention on low resource and energy requirements, as the main targets are single node embedded systems for industrial applications.

1.3 Structure

This section gives an overview of the structure of the thesis. Chapter 2 presents existing work, done in the area of failure prediction and acceleration of related algorithms. Chapter 3 provides a short overview of the prediction model to be accelerated and presents the results obtained by using the prediction model in a real case scenario. This is discussed in detail in the reference work [24]. Chapter 4 focuses on the sequence prediction algorithm and provides an analysis of available parallelism. In Chapter 5 the design of an accelerator is proposed, and in Chapter 6 results in terms of speedup and accuracy are presented. Chapter 7 concludes the work by presenting the contributions and outlining possible future work related to the thesis. The appendix contains a list of symbols used throughout the thesis. Also a list of acronyms is provided.

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 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, 29, 9] as well as the technique described in the main reference [24] itself. [17] describes an error-frequency based approach, [29] uses a data mining approach, and [9] is based on classification. [24] proposes a technique based on error-event sequence processing and delivers the best results with the cost of higher computation time (cf. Chapter 3 for more details).

More recent work uses Hardware (HW) counters of a general purpose Central Processing Unit (CPU) and combines them with Software (SW) instrumentation to analyse failures of single processes (e.g grep, flex, sed) [33]. As industry heads more and more towards cloud computing, it has been proposed to use information of interaction between nodes (instead of analysing single nodes) in order to analyse and predict failures of a distributed system [26, 20].

2.2 Accelerator

The main goal of this master's thesis is to accelerate an extension of the forward algorithm. The forward algorithm is used in the context of Hidden Markov Models (HMMs). It is used to compute the probability of occurrence of a certain sequence, given a HMM. A proposal for a Graphics Processing Unit (GPU) based accelerator for the classic forward algorithm is described in [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, 21] describe a Field Programmable Gate Array (FPGA) based accelerator for protein sequence HMM search, and [30] describes i.a. an approach to accelerate the Viterbi algorithm from the HMMER library[1] using GPUs.

[1] http://hmmer.janelia.org/

Focusing on a more general approach for acceleration, [13] proposes an implementation on FPGA of a parallel floating-point accumulation, and [32] describes the implementation of a vector processor on FPGA.

Quite some research has been done on the question of what type of technology should be used to accelerate computation intensive 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. While [24] proposes to perform the training off-line with no time constraints, it may be beneficial to constantly train the model parameters on-line, due to possible changes of the system. Similar work has been done by accelerating Support Vector Machines (SVMs): [4] describes a FPGA based accelerator for the SVM-Sequential Minimal Optimization (SMO) 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

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

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

- N: number of states of the Hidden Semi-Markov Model (HSMM)
- M: number of observation symbols (size of the alphabet)
- L: observation sequence length
- R: number of Cumulative Distribution Functions (CDFs) (kernels)

These are fixed parameters because they need to be set at design time. They are based on the properties of the system which emits the error events.

The proposed failure prediction method aims to find sequences of error events that led to failures in the system on previous occasions. In addition to a specific sequence of events also the time of occurrence of the event is taken into account (or more precisely, the delay as defined in Equation 3.1). The error events that are produced by the system usually need to be preprocessed and brought in to a form understandable for the predictor. The error events must build a set of distinct symbols, denoted as $\boldsymbol{o} = \{o_1, \ldots, o_M\}$. This set is also called alphabet of size M. Details on preprocessing the events can be found in Section 3.1. The delay of the event at time t_k with respect to the event at time t_{k-1} is described as

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

To detect specific sequences of events, by also taking the time of occurrence into account, an extension of the Hidden Markov Model (HMM) is proposed. The resulting model is called Hidden Semi-Markov Model (HSMM). An extended forward algorithm is used to compute the likelihood of occurrence of a certain sequence of observation symbols, also considering the time of occurrence of an observation symbol. Section 3.3 presents the formal definitions of the algorithm on how the sequences are processed. The computed likelihood of a model is then used in a classifier to decide if a failure is predicted. The decision to predict a failure is based on multiple models where one model computes the likelihood of non-failure sequences. Section 3.4 describes the classification.

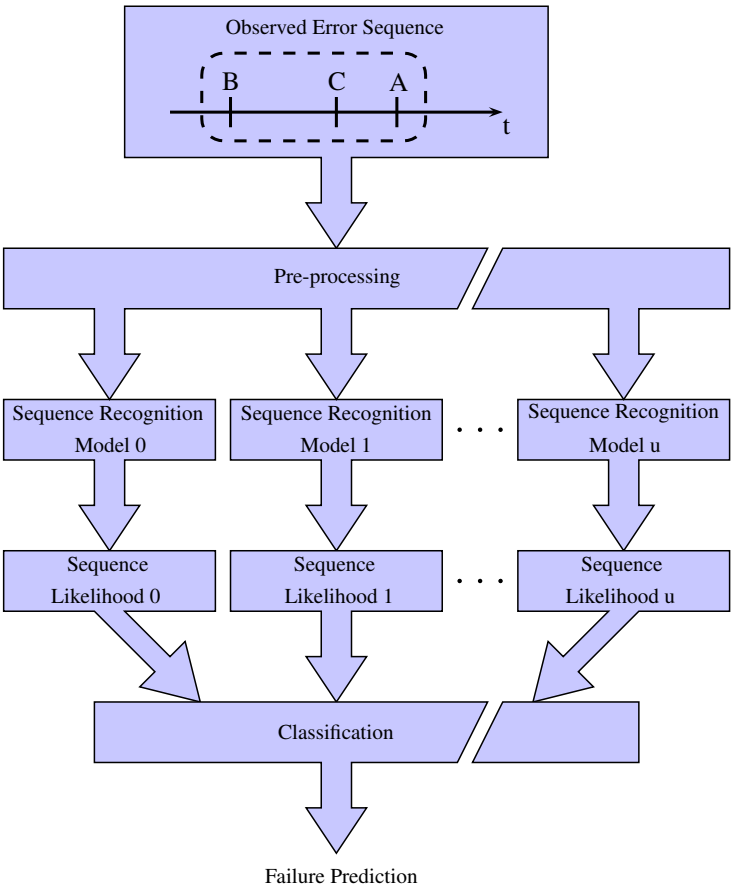

Figure 3.1. Overview of the failure prediction approach. Model 0 is trained with non-failure sequences.

In order to predict failure types with sequence processing, the HSMM has to be trained. The training is performed with an adaptation of the Baum-Welch algorithm. Section 3.2 provides a very brief description and lists the trained features necessary for the sequence processing.

The system supports multiple failure types where for each type a separate model must be built. Additionally a non-failure model is built in order to classify sequences. Figure 3.1 provides a schematic overview of a complete prediction model with multiple failure types.

3.1 Data Processing

The failure prediction method is based on error events. The most common form, how to save error events, are log entries in a file structure. Although work has been done to propose standardized techniques on how log events should be stored ([27, 34]), it is still not common to follow such approaches. This results in a large number of different log structures that makes automated processing of such data difficult. Common issues that need to be taken care of are: different representation of time stamps, numbers in log messages that need to be filtered (e.g.

process id), typos, and different text messages with the same meaning.

The reference work [24] describes methods on how to process log files and create unique error ids that then build an alphabet of symbols. This has been done on real case data. In this work no Hardware (HW) or Software (SW) solution is provided to preprocess data. It is assumed that the events are already in a form that is readable by the accelerator.

3.2 Training of the Model

The model training is based on an extension of the Baum-Welch Algorithm. The model needs to be trained only once. Afterwards it can be used without training. Therefore, this part will not be accelerated. Hence, only a brief, non-formal description of the algorithm is presented. For a more detailed explanation and proof, refer to [24]. The training is used to estimate the parameters for each HSMM.

The parameters to be trained are used by the extended forward algorithm, an algorithm that allows to compute a sequence likelihood for each sequence of observation symbols (error events). The forward algorithm is described in the next section. Following the parameters:

- π_i, forming the initial state probability vector $\boldsymbol{\pi}$ 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, with $i, j \in \{1, \ldots, N\}$
- $\theta_{ij,r}$, the parameters of the kernel r, with $i, j \in \{1, \ldots, N\}$

The estimation of the parameters can be done by using real error event logs associated to real failures. The delay and id of each error event as well as the timestamp and type of the failure are known. With an iterative approach (Baum-Welch algorithm), the parameters can be estimated in order to allow the HSMM to produce a value that describes the likelihood of a specific event sequence to produce a failure of a specific type. Due to the proposed extension, also taking the delay of an event into account, the Baum-Welch algorithm was extended with a gradient descent approach to estimate the kernel parameters of the chosen CDFs.

3.3 Sequence Processing

The following description will provide a complete blueprint of the extended forward algorithm without any explanations or proofs related to the formulation. Those are provided in [24]. While the basic forward algorithm is used to calculate a sequence likelihood based on an HMM, with the extension it is possible to do the same for the proposed HSMM. The extension takes into account that events are not emitted with a constant period. This variance of the delay (cf. 3.1) is modelled by introducing one or more CDFs, also called kernels. The extended forward algorithm is defined as follows:

$$\alpha_1(i) = \pi_i b_{s_i}(O_1) \tag{3.2}$$

$$\alpha_k(j) = \sum_{i=1}^{N} \alpha_{k-1}(i) v_{ij}(d_k) b_{s_j}(O_k); \quad 2 \leq k \leq L \tag{3.3}$$

where $\alpha_k(i)$ is a component of the forward variable vector. $v_{ij}(d_k)$ is called the transition probability, forming the matrix V of size $N \times N$. It is defined as:

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

with

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

$d_{ij}(d_k)$ is forming the matrix of cumulative transition duration distribution functions $D(d_k)$ of size $N \times N$. While the kernel parameters are fixed, with the training of the model, the delay of the event is not. This implies that for each event, a new transition probability matrix must be computed.

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

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

The kernel parameters are dependent on the choice of CDFs. E.g. the Gaussian CDF results in the kernel parameters μ_{ij} and σ_{ij}. A CDF is chosen at design time.

Comparing the extended forward algorithm to the basic version, the only difference is the transition probability matrix. While the extension has variable transition probabilities, those of the basic version are constant. The basic forward algorithm is defined as follows:

$$\alpha_1(i) = \pi_i b_{s_i}(O_1) \tag{3.7}$$

$$\alpha_k(j) = \sum_{i=1}^{N} \alpha_{k-1}(i) a_{ij} b_{s_j}(O_k); \quad 2 \leq k \leq L \tag{3.8}$$

Throughout the document, one transition probability of the basic forward algorithm will always be denoted as a_{ij} and the matrix as A, while for the extension the notation v_{ij} for the probability and V for the matrix will be used. If both versions are addressed at the same time, the notation q_{ij} for one element will be used and Q for the matrix.

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:

$$P_s(\mathbf{o}|\lambda) = \sum_{i=1}^{N} \alpha_L(i) \tag{3.9}$$

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:

$$\alpha_k(i) = c_k \alpha_k(i) \tag{3.10}$$

with

$$c_k = \frac{1}{\sum_{i=1}^{N} \alpha_k(i)} \tag{3.11}$$

By applying scaling, instead of the sequence likelihood (Equation 3.9), the sequence log-likelihood must be computed:

$$\log(P_s(\boldsymbol{o}|\lambda)) = -\sum_{k=1}^{L} \log c_k \tag{3.12}$$

where $\lambda = \{\pi, P, B, D(d_k)\}$, defining the set of parameters.

3.4 Classification

The classification step is based on Bayes decision theory and a classification by threshold. Due to multiple reasons explained in [24], it is not possible to perform a multi-class classification in case of the log-likelihood (which is used if scaling is applied). Therefore the classification problem is reduced to a dual-class problem by selecting the maximum log sequence likelihood of the failure models and comparing it to the log sequence likelihood of the non-failure model:

$$\text{class}(s) = F \iff \max_{i=1}^{u} \left[\log P(\boldsymbol{s}|\lambda_i) \right] - \log P(\boldsymbol{s}|\lambda_0) > \log \theta_{th} \tag{3.13}$$

with $\boldsymbol{s}$ denoting the sequence, λ_i representing the parameter set of the failure models, and λ_0 the parameter set of the non-failure model. θ_{th} is defined as

$$\theta_{th} = \frac{(r_{\bar{F}F} - r_{\bar{F}\bar{F}})P(c_{\bar{F}})}{(r_{F\bar{F}} - r_{FF})P(c_F)} \tag{3.14}$$

To calculate θ_{th}, 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

If $\text{class}(s) = F$, it is predicted that a failure will occur.

If no scaling is used, the dual-class classification problem can be formulated as follows:

$$\frac{P(s|\lambda_F)}{P(s|\lambda_{\bar{F}})} > \theta_{th} \tag{3.15}$$

with $\boldsymbol{s}$ denoting the sequence, λ_F representing the parameter set of the failure model, and $\lambda_{\bar{F}}$ the parameter set of the non-failure model. θ_{th} is defined with Equation 3.14.

For multi-class classification the following rule can be used:

$$\sum_t r_{tF_a} P(\boldsymbol{s}|c_t) P(c_t) > \sum_t r_{t\bar{F}} P(\boldsymbol{s}|c_t) P(c_t) \tag{3.16}$$

Each element r_{tF_a} of the risk matrix defines the cost / risk of associating a pattern $\boldsymbol{s}$ with the class c_{F_a} where it really belongs to class c_t. c_t denotes a failure or the non-failure class, c_{F_a} a failure class, and $c_{\bar{F}}$ the non-failure class.

3.5 Metrics

The failure prediction algorithm as briefly described in this chapter has been verified on real case data. The results in Figure 3.2 are cited from [24] and show the precision, recall, F-measure, and false positive rate metrics of the following failure prediction algorithms:

periodic This is a failure prediction method that is used to estimate a lower bound and is derived directly from reliability theory.

DFT The Dispersion Frame Technique (DFT) is described in [17]. It is based on the notion that errors occur more frequently before failures occur. It is an error-frequency based approach.

Eventset The idea of this method is to find a good set of error events in order to capture as many failures as possible. Training is used. The method is described in [29]. It is a data-mining approach.

SVD-SVM Singular Value Decomposition (SVD) is used to reduce the number of dimensions in a bag of words error sequence representation. This method follows a classification approach. Support Vector Machines (SVMs) are used as classification technique. It is described in [9].

HSMM This is the proposed algorithm of the reference work [24] where it is described in detail. It is the method for which an accelerator is designed in this thesis.

Precision p, recall r, F-measure F_α, and false positive rate fpr are defined as

$$\text{Precision} \quad p := \frac{\text{true positives}}{\text{true positives} + \text{false positives}} = \frac{\text{correct warnings}}{\text{all warnings}} \tag{3.17}$$

$$\text{Recall} \quad r := \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} = \frac{\text{correct warnings}}{\text{failures}} \tag{3.18}$$

$$\text{F-measure} \quad F_\alpha := \frac{p \cdot r}{(1-\alpha)p + \alpha r} \tag{3.19}$$

where α is used to weight the precision.

$$\text{false positive rate} \quad fpr := \frac{\text{false positives}}{\text{false positives} + \text{true negatives}} = \frac{\text{false warnings}}{\text{non-failures}} \tag{3.20}$$

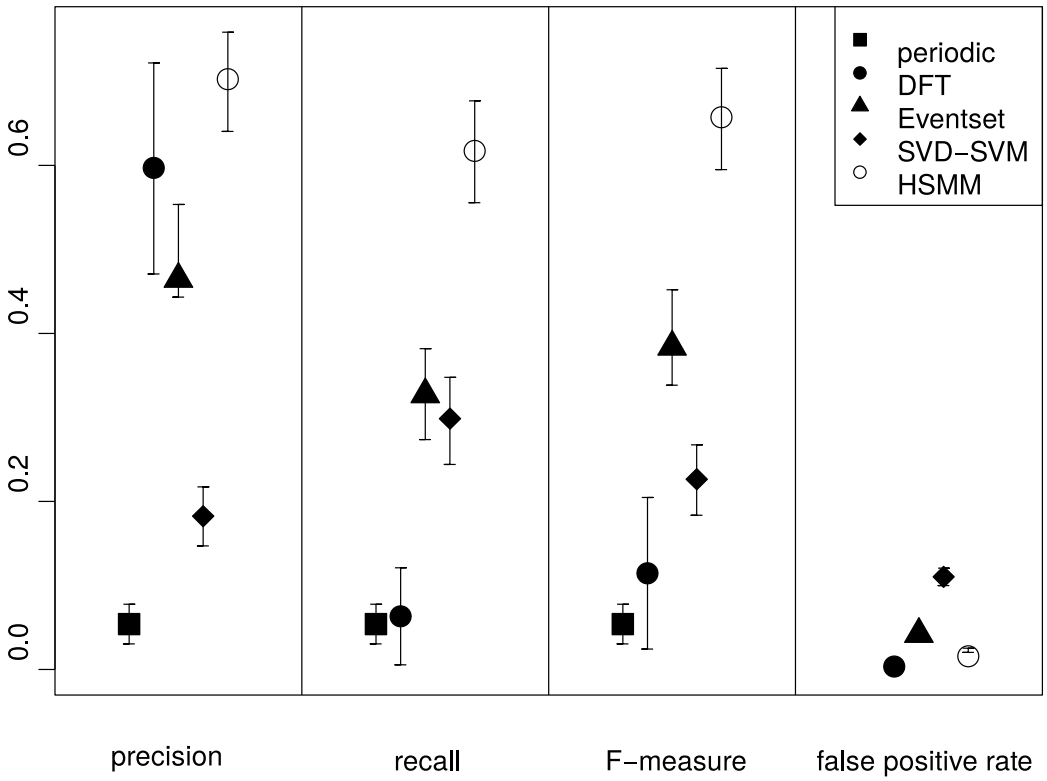

Figure 3.2. Comparison of different failure prediction techniques. The figure was taken from [24].

Chapter 4

Theoretical Analysis of the Forward Algorithm

This chapter provides details about the forward algorithm and available (and useful) parallelization techniques applicable to the algorithm. The generally known forward algorithm as well as the extended version proposed in [24] are discussed. Further, scaling techniques of the forward variables and their impact on data representation choices are presented. Finally, the observations are presented in an overview, and an appropriate choice on possible acceleration Hardware (HW) is made.

4.1 Serial Implementation and Complexity

The sequential implementation of the basic forward algorithm is represented in Figure 4.1. It consist of three steps: the initialization step, the computation of consecutive forward variables, and the final step where the likelihood is computed. The initial forward variable vector α_1 is computed by multiplying, element-wise, the initial state probability vector π with the emission probability vector of the first observation symbol of the sequence $B(O_1)$ (cf. Equation 3.2). The computation of the following forward variables $\alpha_k,\ k \in \{2,\ldots,L\}$ consists of three nested loops: the outer loop iterates over the L sets of N $\alpha_k(i)$ variables where each variable $\alpha_k(i)$ depends on all prior computed variables α_{k-1} and the k-th observation symbol of a sequence. The first inner loop iterates over the N $\alpha_k(i)$ variables of one set where each variable is computed with the innermost loop. The two nested inner loops form the matrix-vector-vector multiplication

$$\alpha_{k+1}(i) = \big(Q\alpha_k\big)(i) \cdot \big(B(O_k)\big)(i) \quad \forall i \in \{1,\ldots,N\} \tag{4.1}$$

where α_k is a vector of size N of the prior computed α variables, Q 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 final step the likelihood is computed, by summing up all elements of the last forward variable α_L (cf. Equation 3.9).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 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
```

Figure 4.1. Pseudo-code of the forward algorithm.

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 implementation of the proposed scaling method is shown in Figure 4.2. The scaling is defined by the Equations 3.10 and 3.11. Due to the scaling, instead of the likelihood, the log-likelihood is computed. Equation 3.12 gives the definition.

The algorithm to compute the sequence likelihood proposed by [24] is an extension of the forward algorithm presented in the Figures 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 previous symbol. The computation of the transition probability matrix Q is implemented with Figure 4.3 and defined by the Equations 3.4, 3.5, 3.6, and 4.6. As in [24], 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 basic as well as th extended algorithm is $O(LN^2)$. The introduction of scaling and/or the extension increases the number of necessary operations by a constant factor. The order of complexity stays therefore the same. This is also true for the space complexity which is of order $O(N^2)$. The order of complexity is important for big N and L. However for smaller values of N and L, the real complexity becomes more important. Sections **??** and **??** will discuss scaling and the extension in detail.

4.2 Parallelism and Feasible Parallelization

Parallelization methods are applied in order to increase the throughput, reduce the latency of a task, or to achieve both at the same time. It can be done at the cost of increased usage of parallel computation units, memory, and memory bandwidth.

Considering only the basic forward algorithm (Figure 4.1), the computation of the likelihood is divided into $L + 1$ 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, with each arrival of a new observation symbol, the last L elements of the observation symbol sequence are used to compute the likelihood (cf. Figure 4.4). 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 system, compared to a non-pipelined, 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 Q and the emission probabilities $b_i(o_k)$ for all steps at the same time. This reduces the load operations by factor L. This is visualized in the Table 4.1. The table shows the pipeline stages with input values that are fed to the stage before the execution and the output values resulting after the execution of the pipeline stage. Figure 4.5 shows a schematic representation of the pipeline. Note that the input values Q 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 they depend on d_k. This will be discussed further when

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 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
```

Figure 4.2. Pseudo-code of the 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
% @param P:         transmission probabilities
% @return V:        extended transition probabilities
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [V] = compute_tp(N, dk, cdf_param, P)
    % compute all elements of v
    for i=1:N,
        for j=1:N,
            V(i, j) = P(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
```

Figure 4.3. Extension of the forward algorithm with only one (Gaussian) kernel.

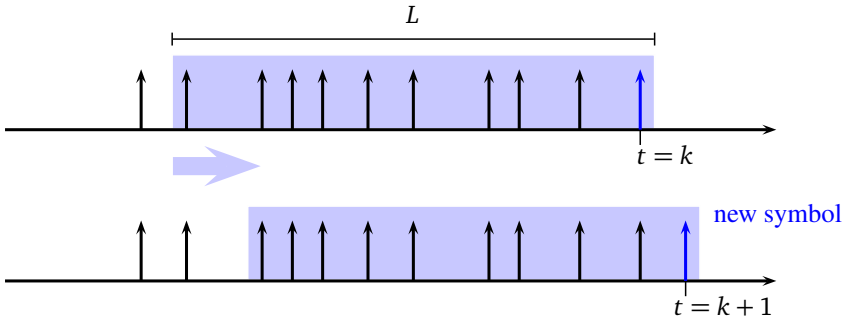

Figure 4.4. Example of a sliding window over an observation sequence of the last $L = 10$ observation symbols.

		Pipeline				
Symb	I/O	Init	Step 2	…	Step L	Final
O_1	in	$B(O_1)$	$B(O_1)$, $Q(d_1)$, 0	…	$B(O_1)$, $Q(d_1)$, 0	0
	out	$\alpha_1(O_1)$	0	…	0	0
O_2	in	$B(O_2)$	$B(O_2)$, $Q(d_2)$, $\alpha_1(O_1)$	…	$B(O_2)$, $Q(d_2)$, 0	0
	out	$\alpha_1(O_2)$	$\alpha_2(O_{1,2})$	…	0	0
⋮		⋮	⋮		⋮	⋮
O_L	in	$B(O_L)$	$B(O_L)$, $Q(d_L)$, $\alpha_1(O_{L-1})$	…	$B(O_L)$, $Q(d_L)$, $\alpha_{L-1}(O_{1,\ldots,L-1})$	0
	out	$\alpha_1(O_L)$	$\alpha_2(O_{L-1,L})$	…	$\alpha_L(O_{1,\ldots,L})$	0
O_{L+1}	in	$B(O_{L+1})$	$B(O_{L+1})$, $Q(d_{L+1})$, $\alpha_1(O_L)$	…	$B(O_{L+1})$, $Q(d_{L+1})$, $\alpha_{L-1}(O_{2,\ldots,L})$	$\alpha_L(O_{1,\ldots,L})$
	out	$\alpha_1(O_{L+1})$	$\alpha_2(O_{L,L+1})$	…	$\alpha_L(O_{2,\ldots,L+1})$	$P_s(O_{1,\ldots,L})$
⋮		⋮	⋮		⋮	⋮

Table 4.1. Tabular representation of the pipelined forward algorithm, with observation symbol O_k and its delay d_k. Here $O_{i,\ldots,k}$ is a short notation for $O_i, \ldots, O_k$.

the extension is considered.

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 first observation symbol $b_i(O_1)$. This can be fully parallelized by replicating the multiplication operation N times. Doing this results in an 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.

The computation of the subsequent forward variables α_k, with $k = 2 \ldots L$ are equivalent in terms of HW. 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 into 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 chequerboard block matrix partitioning (decomposition in rectangular sets of elements). These partitioning methods are shown in Figure 4.6. The number of subsets must correspond to the number of available com-

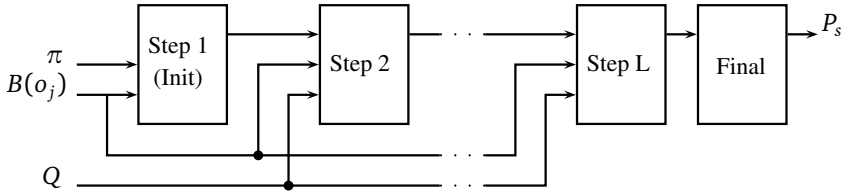

Figure 4.5. Schematic representation of the pipelined forward algorithm.

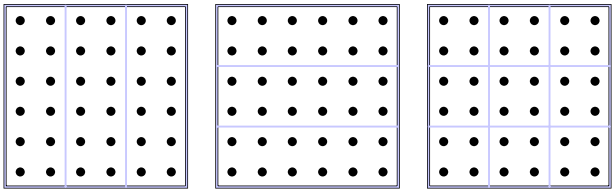

Figure 4.6. Matrix partitioning (from left to right): column-block-striped, row-block-striped, and chequerboard blocks.

putational 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 with the emission probability vector. This is again the same case as the initial step described above. In case of the block-striped matrix partitioning, the maximally achievable increase of the throughput is a factor of N. The latency can be decreased by factor $\frac{1}{N}$. This is possible, 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 chequerboard partitioning yields a lower gain but may be considered, in case of fixed computational unit architecture (e.g. Central Processing Unit (CPU), Graphics Processing Unit (GPU)), in order to increase the utilization of resources 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 accumulating the values sequentially, N units can be used to first multiply two operands together, then adding $\frac{N}{2}$ resulting operand pairs in a second step, and then consecutively adding resulting pairs until only one value is left. This process is visualized in Figure 4.7. The maximal increase of throughput is of factor $\log_2(N)$. The latency can be decreased by factor $\frac{1}{\log_2(N)}$. This is possible, 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 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 of $log_2(N)$ and $\frac{1}{log_2(N)}$ respectively as described above.

The following sections will describe the impact on performance if scaling or the extension of

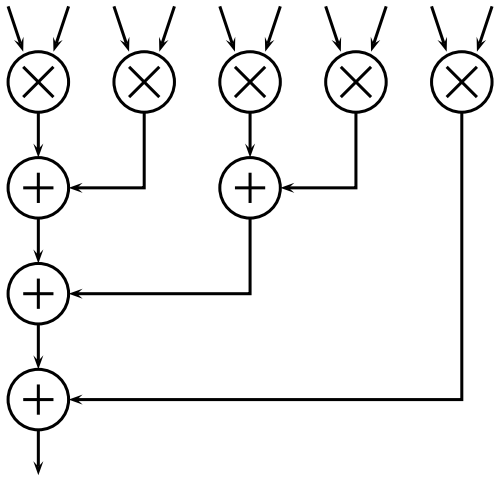

Figure 4.7. Example of a reduction tree with $N = 5$.

the forward algorithm is implemented. Also the availability of parallelization in both cases will be discussed. Finally an overview of available parallelism and a discussion about the usefulness of each parallelization method in the context of the different algorithm implementations will be provided.

4.3 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). Both are of the same order $O(N^2)$. However, the introduction of scaling may increase the usage of resources 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 resources. Additionally, instead of the sequence likelihood (Equation 3.9) the sequence log-likelihood (Equation 3.12) 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 analysing

the operands, an average scaling factor can be computed. Using the knowledge that all the operands are probabilities,

$$\begin{aligned} \hat{\pi} &:= \frac{1}{N}\sum_{i=1}^{N}\pi_i = \frac{1}{N} \\ \hat{q} &:= \frac{1}{N}\sum_{j=1}^{N}q_{ij} = \frac{1}{N} \\ \hat{b} &:= \frac{1}{M}\sum_{j=1}^{M}b_{ij} = \frac{1}{M} \end{aligned} \tag{4.2}$$

and doing the computation of the forward variables,

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

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 too much or too 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 the biggest value of a specific event in B (scale $\max\big(B(o_j)\big)$). By doing this, the scaling factor will be too 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 the on one hand on the necessary precision and on the other hand on the choice of accelerator type. While general purpose HW such as a CPU, GPU or Digital Signal Processor (DSP) (to some degree) offer an abstraction to make the representation type transparent to the developer, specialized HW such as a Field Programmable Gate Array (FPGA) or Application Specific Integrated Circuit (ASIC) offer no such abstraction. For the latter devices, floating-point operations increase the complexity of the HW design and the necessary HW resources considerably. In terms of performance, general purpose devices also benefit from a sparse usage of floating-point values. The complexity of the Software (SW) development of

general purpose devices 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. This implies that by omitting scaling to save resources, a floating-point representation must be used. This, in turn, 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 analysed in more detail in Chapter 5, when the technology of the accelerator has been chosen.

4.4 Extension of the Forward Algorithm

The proposed extension uses a non constant transition probability matrix. For every occurring 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 3.4 and 3.5). Computing N^2 cumulative distribution functions is time intensive, but it only needs to be computed once per d_k and can then be stored for later use. 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 4.4). 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 4.5 and Table 4.1). The computation of the transition probability matrix can be fully parallelized with $R*N^2$ computation units to calculate a Cumulative Distribution Function (CDF) 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. E.g. 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 more complex than the simple Multiply-Accumulate (MACC) operation necessary to calculate a forward variable (cf. the list below). Due to this, the computational units calculating the transition probability matrix must provide 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.

Laplace 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 order, the Laplace distribution is equivalent to 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. The Gaussian kernel is defined as:

$$\kappa_{ij,gauss}(d_k|\mu_{ij},\sigma_{ij}) = \frac{1}{\sigma_{ij}\sqrt{2\pi}}\exp(-\frac{(x-\mu)^2}{2\sigma^2}) \tag{4.6}$$

Using the Taylor expansion and the substitution $x = \frac{d_k-\mu}{\sigma}$, the CDF can be expressed as

$$\Phi(x) = \frac{1}{2} + \frac{1}{\sqrt{2\pi}} \cdot \exp(-\frac{x^2}{2}) \cdot \left[x + \frac{x^3}{3} + \frac{x^5}{3\cdot 5} + \cdots + \frac{x^{2n+1}}{3\cdot 5\cdots(2n+1)} \right] \tag{4.7}$$

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, in order to not reduce the throughput of the fully parallelized computation of the forward variables, a small pipeline of two stages must be built. 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. Figure 4.3) can be maximally parallelized by using N reduction trees to compute the sum of rows and N subtracters to correct the diagonal elements.

4.5 Parallelization Options and Scalability

In the sections above, a lot of parallelization options have been proposed. A maximal parallelization can hardly be achieved due to the immense requirement of resources 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 combinations is done. Then the results will be discussed, and a choice will be made. Finally some conclusions about a reasonable

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.

amount of applicable parallelization options will be drawn in respect to the chosen architecture. Table 4.2 shows the pipelined architecture (cf. Figure 4.5), the parallel architecture in case of maximal row partitioning (cf. Figure 4.6), the combination of both architectures, and in the last column the combination of both architectures plus the reduction tree (cf. Figure 4.7). The basic and the extended forward algorithm are both of the same complexity order. As only the complexity order is considered, different computation times of different operations as well as scaling operations can be ignored. For simplification reasons, 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 respectively 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 too 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 system where lots of people depend upon (e.g. 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 bottlenecks and have a negative 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 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 whether 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 too

		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).

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 choose the pipelined architecture would be a simple state transition model (cf. Section 4.7). It 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 too 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 resources, 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.

4.6 Selection of Accelerator Type

In order to choose an appropriate accelerator type, first a list with different acceleration HW is presented. For each type the most common benefits and shortcomings are listed. 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 (CPU) falls into the category of General Purpose Processors (GPPs) and is (usually) of the type Single Instruction, Single Data Stream (SISD). It is very flexible in terms of SW 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 HW 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 4 GHz). CPUs come with the drawback of a high power consumption, limited parallelization options (small number of cores), and a fixed HW architecture that causes big computation overheads (instruction pipeline, memory hierarchy, generalized computation units).

GPU

Like the CPU, the Graphics Processing Unit (GPU) still falls into the class of GPPs due to the HW abstraction layers. A GPU is composed of a lot of small but quite powerful streaming processors of the type Single Instruction, Multiple Data Streams (SIMD). This processing power allows a lot of parallelization at a low price. As the CPU, also the GPU is very flexible in terms of SW interpretation. However it requires some HW understanding in order to use the parallel power in an optimal way. A GPU offers 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 HW abstraction layer provides flexibility in SW and ease of use, it is the main reason for a computation overhead for basic operations. The fixed HW architecture (especially the memory hierarchy) can prove to be a drawback in specific cases (e.g. if data usage always provokes cache misses).

DSP

The Digital Signal Processor (DSP) is a specialized integrated circuit, largely used for digital signal processing. The key components of DSPs are optimized MACC instructions, special SIMD operations, and for DSP operation optimized memory architecture. They provide a HW 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 (FPGA) is a customizable integrated circuit. It provides large resources of logic gates as well as standard blocks such as RAM or highly optimized MACC units. FPGAs provide high performance for very specific designs, 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 HW representation of the function. This direct representation allows a very low overhead as operations are done directly in HW without any instruction pipeline. FPGAs provide a high amount of HW

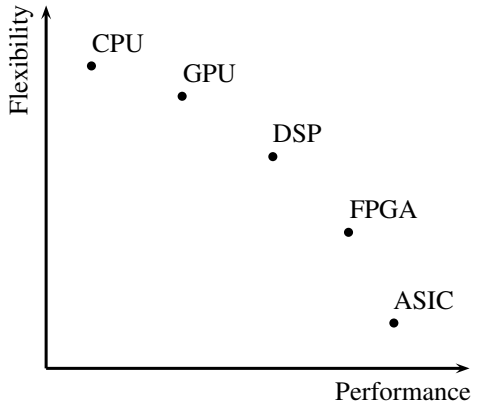

Figure 4.8. Approximate representation of flexibility versus performance of HW devices.

flexibility that is only topped 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 to 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 HW 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 deep knowledge of HW is necessary to successfully implement a function on FPGA.

ASIC

The Application Specific Integrated Circuit (ASIC) is, as the name indicates, 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 achievable with today's HW knowledge. An ASIC has ideally no overhead as the HW 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. Deep HW 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 a better performance - power consumption ratio always comes along with a lower the flexibility of the device to accept general function descriptions (cf. Figure 4.8, e.g. 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 is computationally 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 are not 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 MACC 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 too much of 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 resources that were not available for this thesis. This leaves the FPGA: An FPGA combines the parallel power of DSPs for MACC operations but adds the possibility to design a specific HW architecture to compute the transition probabilities needed for the extension. While the performance will not be at a level of an ASIC, the specialized HW will still outperform any other device because of the customizable memory interface (even at the lower frequencies of FPGAs).

4.7 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 in the Hidden Semi-Markov Model (HSMM). This is not necessarily the case, as it is often possible 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 dependent on the architecture and will hence be discussed in the next chapter where the focus lies on one specific type of HW device. 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 any more as a lot of computational units would be idle in most of the time. A sparse matrix vector multiplication method should be used in this case, but this will not be analysed in this thesis.

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 Field Programmable Gate Array (FPGA).

To design the accelerator the top-down approach was applied: the algorithm was broken down into blocks where each of them is broken down further until the basic functional blocks of the FPGA can be used for the implementation. The implementation then follows 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" respectively, 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 and memory controllers (FLASH CTRL and RAM CTRL, respectively) 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 whether the present sequence leads to a failure or not and returns this result as a boolean value. In terms of Hardware (HW), 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 design details of the failure blocks are presented. The functionality of the pipelined architecture has already been described in Chapter 4.2. Figure 4.5 depicts the basic schematics. The same architecture is shown in Figure 5.2 with a simplified Register Transfer Level (RTL) schematic. The main aspect of the chosen design is to use the high performance Multiply-Accumulate (MACC) units available in modern FPGAs. These so called Digital Signal Processor (DSP) slices allow to perform, besides other functions, fully pipelined MACC operations at frequencies of up to 450 MHz. The manual [31] describes the vast functional possibilities of the DSP slices available in the series 7 FPGAs of Xilinx[1]. As shown in Figure

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

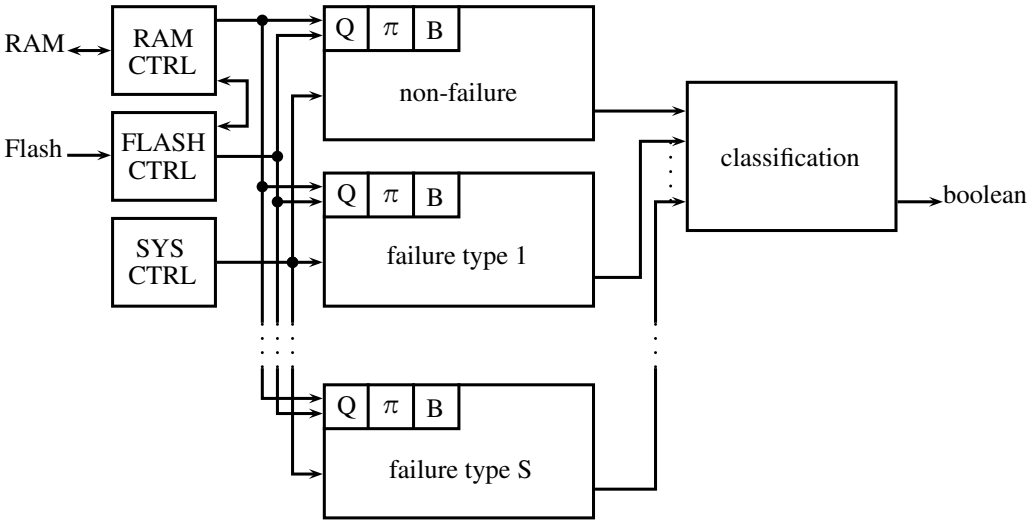

Figure 5.1. Schematic representation of the top architecture of the failure prediction algorithm.

5.2, in each pipeline stage the transition probabilities (coming either from memory in case of the basic forward algorithm or from a computational unit in case of the extended algorithm) are multiplied with the forward variables (read out of the First In, First Out (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 MACCoperation and two stages for the multiply operation. It takes $N+3$ cycles to accumulate all multiplication pairs ($\alpha_{k-1,j} * q_{i,j}$) and 2 cycles to perform the final multiplication. The setup time of three and two cycles only needs to be considered once if a well timed reset of the accumulator is performed. Therefore the computation of all components of the vector α_k adds up to N^2+5 cycles.

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 per $N+3$ cycles and computes either non-valid data or is idle (if disabled) during the rest of the time. This is a very poor utilization of resources and can be optimized by reusing the first multiplier to perform the second multiplication. Doing this increases the necessary cycles to a minimum of $N*(N+3)+2$ cycles but reduces the number of required 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 in the 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

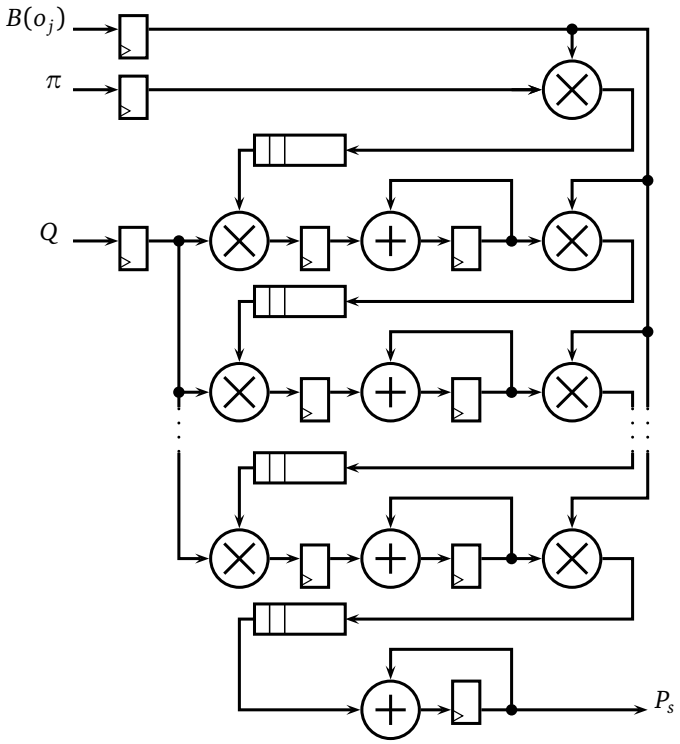

Figure 5.2. Simplified RTL representation of the pipelined forward algorithm.

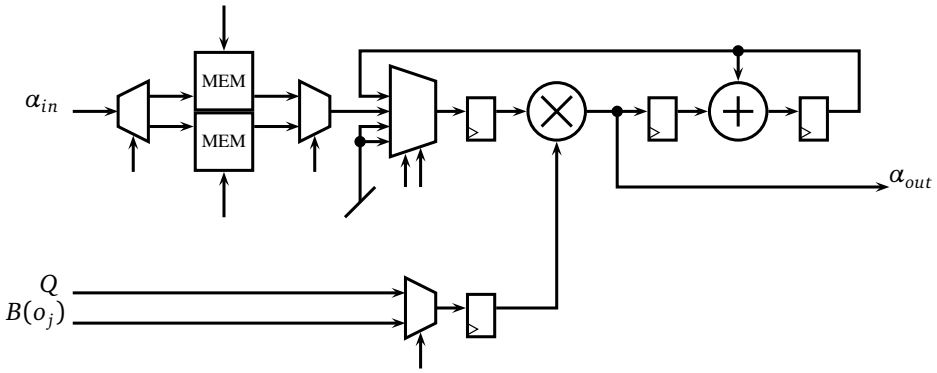

Figure 5.3. RTL implementation of a pipeline stage with dual memory queue and reuse of components.

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 value. 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 (as discussed in Chapter 4.7), 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 will not 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 resources (see Chapter 5.4 for more information).

Figure 5.3 shows the detailed design of one pipeline stage, if the multiplication unit (DSP slice) is reused and if two memory blocks are used (alternating one to write new data and one to read data stored from the last α vector computation). The control signals are not labelled here for reasons of readability and will be explained in detail in Chapter 5.5.

5.2 Extension

If the extension is used, for each new event, the transition probability matrix must be computed. This computation is described by Equation 3.4 and the corresponding Cumulative Distribution Function (CDF) definition. The serial implementation is shown in Figure 4.3. In order to not influence the throughput with the extension, a top level pipeline of two stages must be introduced. In the first stage the transition probability matrix is computed and in the second stage the forward variables. As both stages need to use the same amount of cycles to complete, the computation of transition probabilities is limited to $N*(N+3)+2$ cycles. This is the amount of cycles available to compute N^2 transition probabilities if the throughput of the system should stay the same. Introducing this pipeline doubles the latency. Figure 5.4 shows the RTL implementation of the transition probabilities. It takes as input the CDF values and the transmission probability values P and performs the correction of the diagonal of the matrix V (as expressed in Equation 3.4). While the product of the CDF and the P values are stored directly into an

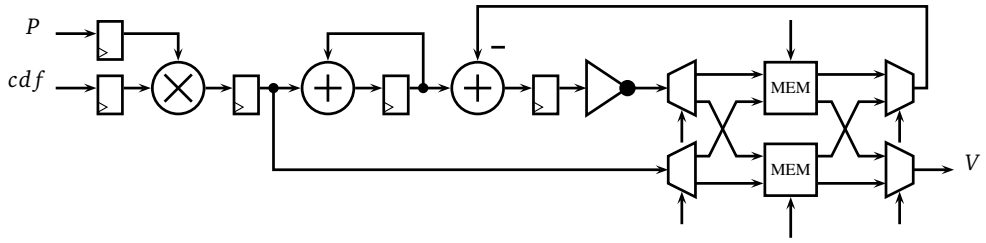

Figure 5.4. RTL implementation of the transition probability computation.

internal memory block, the elements are accumulated. After N cycles the diagonal value corresponding to the row that was just accumulated ($v_{i,sum} = \sum_{\substack{h=1 \\ h \neq i}}^{N} v_{ih}$) is subtracted, and a bitwise inverse is performed to compute $1 - v_{i,sum}$. In order to not interrupt the pipeline flow when a corrected diagonal value must be written, a dual-port memory is needed. Note that the values need to be written column-wise (to use a pipelined accumulator), but to calculate the α vector, they need to be read row-wise. To prevent interference, two memory blocks are necessary that act alternating as target for storing values and as reading memory (similar to the α memory queues described in the previous section).

Figure 5.5 shows the design of the exponential CDF (cf. Equation 4.4), and 5.6 shows the design of the Laplace CDF (cf. Equation 4.5). Due to the very complex computation of the Gaussian CDF (cf. Equation 4.7) the design of this CDF was omitted. As described in [23], the occurrence of failure events can often be described as a Poisson process. This justifies the choice of omitting the Gaussian CDF for this application. All CDFs need the exponential function. In addition, the Laplace CDF needs a division operation, an adder, and a comparator. The shift operation is constant and can be achieved with fixed wiring. The exponential CDF only needs a multiplier and a comparator. Of the required components, the exponential function poses the biggest problem. Fortunately, implementation solutions of exponential functions for FPGAs have been proposed ([22, 8]). These solutions are however designs for floating-point numbers. Therefore, the division, multiplication, comparison, and addition operations must also be done in floating-points. It is out of the scope of this thesis to design and/or implement these elements. There is however the Xilinx CORE Generator tool[2] that can build a functional blocks performing floating-point operations. Unfortunately there is no support for exponential functions. Another option of solving this problem is to use a Look Up Table (LUT) storing results of a precomputed exponential function. The range of the computed values depends on the trained features of the chosen CDF. With an interpolation, intermediate values are calculated. The benefit of this option is the possibility to go back to a fixed-point representation for all operations. To precompute the complete CDF would be challenging, due to the fact that for each value in V specific CDF parameters are required and hence imply an increased memory complexity to the order $O(C * N^2)$, with C denoting the number of values necessary to describe the CDF. The extension will not be implemented on a HW solution within the scope of this thesis.

The design of both CDFs as well as the diagonal correction of the matrix V is fully pipelined. It takes $N^2 + C$ cycles to compute the correct matrix V where C is the pipeline length. The

[2] http://www.xilinx.com/tools/coregen.htm

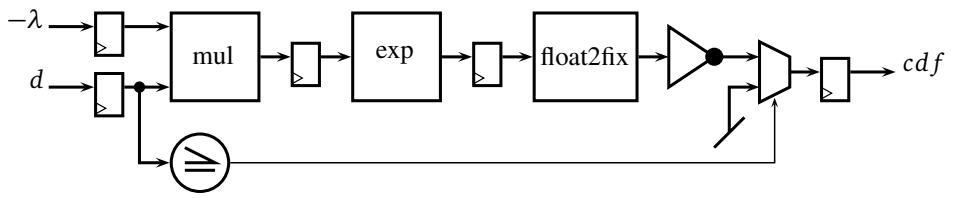

Figure 5.5. RTL implementation of the exponential CDF computation.

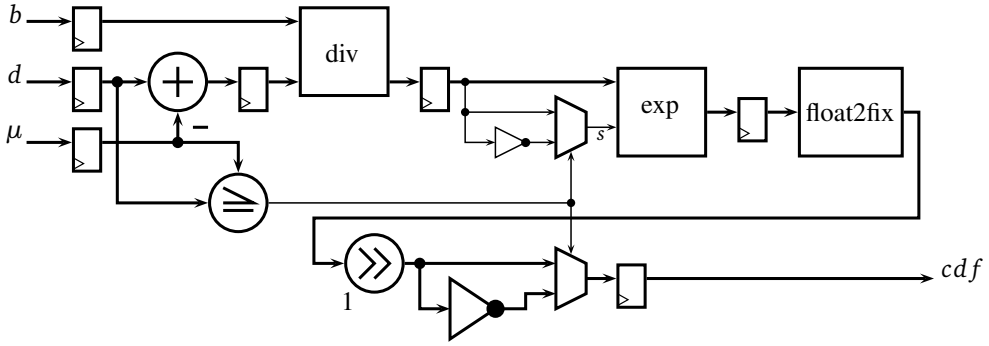

Figure 5.6. RTL implementation of the Laplace CDF computation.

number of cycles needed for the forward variable computation are $N*(N+3)+2$. For $C \leq 3N+2$, the throughput of the system is not affected. As the Xilinx CORE Generator allows to configure the latency of each block (lower latency means more resources), the design can be optimized for a specific N.

This design proposal of the extension is by far not as optimized as the design proposed for the forward variable computation. A pure fixed-point design would allow to reduce the resources usage. This would allow to propose a reasonable design for the Gaussian CDF (a CDF which is very often used).

5.3 Operand Scaling and Precision

The DSP slices used to multiply and accumulate operations are designed for a fixed-point data representation, hence they 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, leads 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 avoided 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 be 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 expo-

nent must be shifted by this difference (and possibly rounded/truncated). This process is called normalization. After the normalization, the mantissas can be added, and finally the resulting value needs to be normalized again. 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. The result must then be truncated/rounded and normalized.

The work [3] presents a solution for floating-point operations, including multiplication and addition, in single precision[3] 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 have to be extended by a parametrizable exponent size. Another issue In [3] is the absence of a optimized and fully pipelined MACC 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 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 [28, 10] discuss the choice of fixed-point versus floating-point representation with respect to DSP devices. This is also applicable for FPGAs. An important point is the trade-off between speed and accuracy. While a fixed-point representation allows to process more data at a lower precision, a floating-point representation allows a higher precision at a lower speed. The optimal choice depends on the application. [10] compares audio with video processing. Audio processing is, due to the sensitivity of the human ear, more prone to errors and demands high precision. In video processing precision is less an issue. However the amount of image data to be processed in case of a video is much bigger compared to a audio stream. By trading speed for more precision, it would make sense to use a floating-point representation in case of audio processing. On the other hand by trading precision for more speed, a fixed-point representation would make sense for video processing. In the application at hand, speed is the most important factor in order to process as much error events as possible (the speed of the system limits the average occurrence frequency of the error-events). Considering this, the choice of a fixed-point representation is reasonable. Another factor is the design time. Due to the complexity of a floating-point data representation, the design time of an application operating with floating-points on FPGA is higher than a fixed-point implementation. Due to the limited amount of time available for this theses, a fixed-point implementation is more realistic.

The DSP slices allow fixed-point MACC operations with an operand of bit width 18 and one of bit width 25. Internally the device works with a bit width of 48 bits. This 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 five additional bits are not necessary in the

[3] IEEE 754-2008

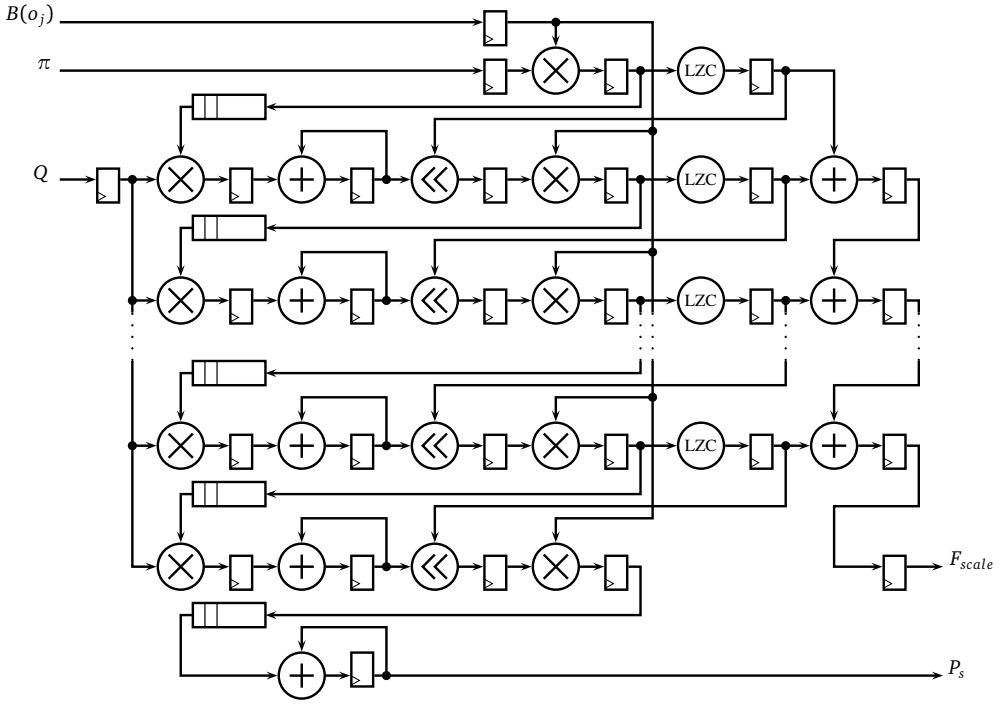

Figure 5.7. Simplified RTL representation of the pipelined forward algorithm with scaling using shifters and leading zero counters.

present case. After the MACC operation, the second multiplication takes again operands of width 18 bits and 25 bits as input. Therefore, the result of the MACC needs to be truncated. In order to not loose too much information, at this step scaling must be introduced.

As already discussed in Chapter 4.3 the scaling method proposed by the reference work should be avoided in this implementation due to the high requirements in terms of resources. 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 (Leading Zero Counter (LZC)) unit is introduced. This unit is purely combinational and counts the leading zeros of an 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 with 2^{LZC}). Also, in this stage (and every following), a LZC unit is added in the same way as described above. The LZC result of each stage is added to the LZC result of the next stage. 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 an overflow in the shifting operation. It is important to use the same value for all operands to being able to accumulate the operands without normalization in the next pipeline stage.

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 5.7.

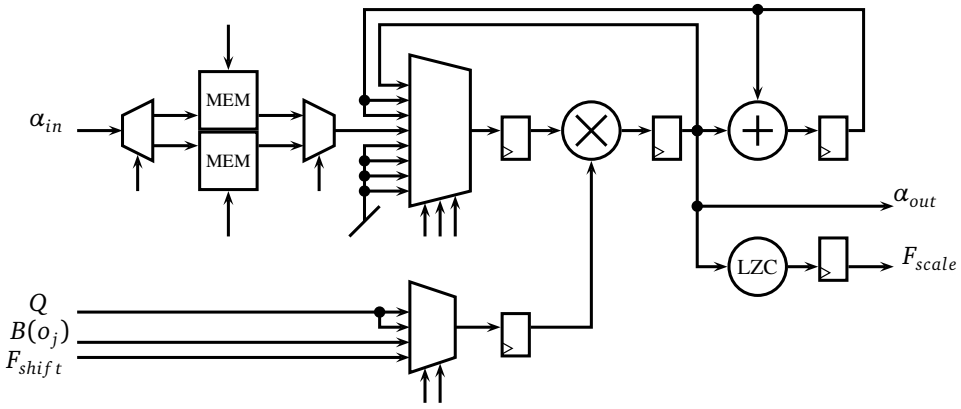

Figure 5.8. RTL implementation of a pipeline stage including scaling.

To perform the scaling, 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 needs to be scaled. The scaling 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 LUT 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 bits corresponding to the width of the lower part shifted before to the right (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. This results in a shifted value of op by LZC bits to the right. All these operations can be done by using only one DSP slice. Figure 5.8 shows the necessary schematics. As described in Section 5.1, the simplified representation of the memory queue has been replaced with two memory blocks.

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 Q can be preprocessed and scaled. Also here, the element with the lowest number of leading zeros decides the scaling factor. By doing this, the assumption that an overflow cannot happen due to the fact that probabilities are involved doesn't hold anymore. Hence, by introducing prescaling, overflow handling and correction must be added.

5.4 Memory Architecture and Management

A huge advantage of FPGAs is the flexibility in terms of memory architecture. In General Purpose Processors (GPPs), usually a memory hierarchy is used (from fast and small to slow and big): registers, 1st, 2nd (sometimes 3rd) level cache, then Random Access Memory (RAM), a hard drive and finally the cloud. Usually it is assumed that the difference in access time between two consecutive levels is one order of magnitude. Such a hierarchy is very efficient if no knowledge about the data flow is available (which is usually the case for GPPs). Sizes of fast caches

are quite small and are only useful if data is frequently reused. In the present case however such a memory hierarchy would be bad because of the high velocity of data. The algorithm at hand always requires the complete matrix Q, which is of size N^2, to compute the next forward variable. The same is true for the vectors π and $B(o_j)$. In case of a memory hierarchy this implies that either the cache is big enough, or there will always be cache misses. In such cases it is much more beneficial to build a scratch pad memory architecture: In this type of architecture, the hierarchy is very flat, and one aims to design blocks of memory of appropriate size to store the necessary data. In FPGAs, such block ram is internally available. This has been used to store the values of π, Q, and B. In case of the extension, instead of Q the CDF parameters and the values of P are stored. The π, Q/(CDF and P) parameter values are stored after they have been trained and are never changed afterwards.

B changes depending on the occurring event. There are usually to many different events ($M = 1000$) to store all possible B matrices inside the FPGA. A two staged pipeline is necessary to keep the throughput at maximum level. In case of the extension this top level pipeline already exists. 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 may 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 by small amounts. 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 too 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 of the accelerator the buffer is designed too 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 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.

In Chapter 5.1, the option of sparse matrix storage methods has been discussed. In [14], several methods on how to store sparse matrices are compared. In the comparison included are the well known techniques Compressed Sparse Row (CSR), Coordinate (COO), and ELLPACK-/ITPACK (ELL). The proposed method takes advantage of the fact that FPGAs can operate on a single bit level. It is shown that storage methods of the type Bit-Vector (BV) can reduce the storage overhead.

While the reduction of storage overhead is a key aspect of sparse matrix storage techniques, another aspect must be considered. In order to not interrupt the pipeline of the proposed design of the accelerator, α vector components corresponding to zero-values in the transition probability matrix need to be skipped. In order to do that, the address of the α vector memory

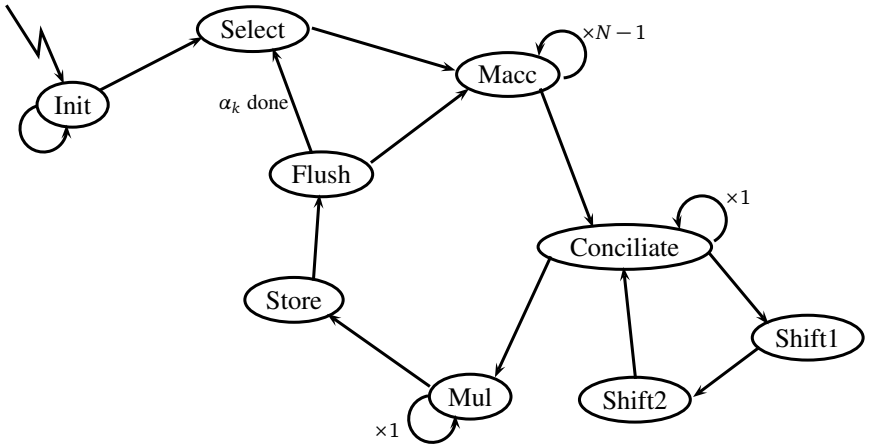

Figure 5.9. Mealy state machine to control the pipeline.

must be increased to the next corresponding non-zero Q value. This can only be achieved in a fast way, if the number of consecutive zero positions is directly obtainable by reading the sparse matrix storage. Considering this and using the results of the work presented in [14], the best option is to use the Compressed Bit-Vector (CBV) or the Compressed Variable-Length Bit-Vector (CVBV) technique. The final choice falls on the CBV technique because the CVBV technique increases the complexity of memory controller with the sole benefit of only small storage overhead reduction. Sparse matrix storage techniques are not further discussed in this thesis.

5.5 Controlling the Pipeline

The pipeline of the forward algorithm design is controlled by a global controller unit. The task of the controller is to balance the pipeline stages. If in a pipeline stage less operations are necessary than in another the computational units are only enabled if a computation is necessary. Idle computational units are disabled in order to minimize dynamic power consumption (prevent the input signals from switching the transistors inside the unit).

The initial stage as well as the final stage need less computation than the $L-1$ intermediate stages. The intermediate stages computing α_k are all equal in terms of HW. The controller is realized with a state machine. Figure 5.9 provides a schematic representation. A detailed list of all the control signals is provided in table B.1.

5.6 Implementation and Testing

The implementation of the algorithm was done following the bottom up approach: Each functional block was tested separately using test scripts and the iSimi[4] simulator of the Xilinx ISE[5] bundle. Figure 5.10 shows an example simulation of the controller.

[4] http://www.xilinx.com/tools/isim.htm
[5] http://www.xilinx.com/products/design-tools/ise-design-suite/

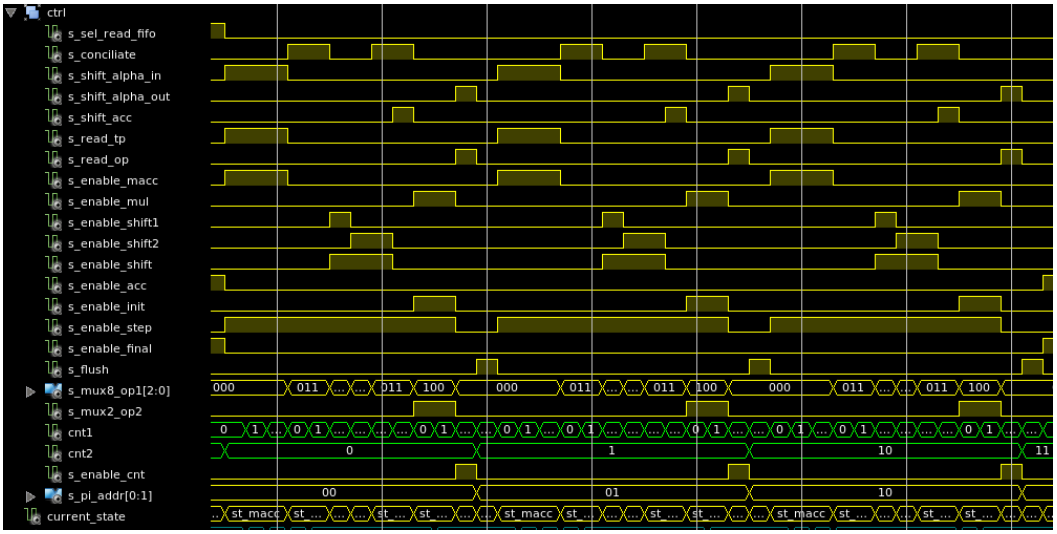

Figure 5.10. Simulation example of the controller.

The implementation has been kept as generic as possible with only one configuration package that allows to set the values N, L, and M as well as the operand widths. This generic options have however no impact on the blocks automatically generated by the Xilinx CORE Generator tool[6]. These need to be generated again if parameters in the configuration package are changed.

A big issue going from simulation to a real implementation on a physical HW board, are timing constraints. In order to operate at a specific frequency, the longest critical path is not allowed to be longer than the period of the clock. Synthesizer options like "register balancing" may help to reduce the critical path, but often there is no way around adding additional registers (and increase the pipeline length and hence the latency) or manually placing critical components in closer proximity to each other. The most constraining unit is the lead zero counter to perform the scaling. Fortunately the scaling factor is not required immediately (cf. 5.7) hence it can be pipelined in order to reduce the critical path. Another critical unit is the controller. Only one controller provides all control signal for the pipeline. This results in a big fanout of the control signals (increasing with L) and long paths to reach each pipeline stage. The problem of the fanout can be solved by using clock buffers, and the long path can be optimized by manually placing the controller in the centre of the device. Another option would be to replicate parts of the controller or find relations between control signals and compute local control signals in the pipeline stages. Such optimizations for higher frequencies have not been done due to the complexity and the limited available time.

The Xilinx CORE Generator has a bug with block RAM generation in case of a initialisation of the RAM with a coe file. Address 0 always returns undefined. This is only a problem in case of initialisation of the RAM with values. In a real application where the RAM initialised with values read from a non-volatile memory, this bug has no impact.

The implementation of the basic forward algorithm with a simple scaling scheme can be

[6] http://www.xilinx.com/tools/coregen.htm

found on github[7].

[7] https://github.com/moiri/USI-thesis-accelerator

Chapter 6

Results

This chapter presents the results achieved by running sample data on the accelerator. Due to a lack of real application data, which is very hard to come by due to confidentiality concerns, test runs have been performed with pseudo random data respecting the properties of Equation 4.2. Random data can be used because [24] performed a thorough evaluation of the prediction algorithm and showed that accurate predictions can be made. In this thesis results in terms of speedup, accuracy, and usage of resources are provided. The tests have been performed on the developer board Nexys4[1] from Digilent. The board is composed of an Artix-7 XC7A100T-1[2] Field Programmable Gate Array (FPGA) from Xilinx and peripherals (e.g. Random Access Memory (RAM), flash, communication interfaces). It is a low price product that offers limited amount of computation units and memory. Tests with values for N and L of up to 100 respectively 50 have been performed on this device. In order to test the design, the internal memory of the FPGA has been loaded with fixed data, and the device was configured to compute the equivalent to the likelihood of one sequence of event occurrences.

In the following, first the accuracy will be discussed. Then the speedup of the accelerator compared to a Central Processing Unit (CPU) is presented, and finally the resource requirements are put in relation with the values of L and N.

6.1 Accuracy

To test the accuracy of the accelerator, the same data set was used to run the algorithm on the accelerator and on a test CPU. The results from both devices were used to calculate an error (cf. Equation 6.1). The result of the CPU was used as reference value. On the test CPU the basic algorithm was computed (without scaling) for feasible values of N and L. The double precision of the floating-points is able to compute a likelihood for $L \leq 106$, assuming $M = 1000$.

$$e = \frac{|result_{cpu} - result_{fpga}|}{result_{cpu}} * 100 \tag{6.1}$$

Table 6.1 shows the accuracy of the accelerator with operand widths of 18 bits (OP_2) for the emission probability vector B and the transition probability matrix Q and 25 bits (OP_1) for the initial probability vector π, the intermediate forward variables α, and the likelihood.

[1] http://www.digilentinc.com/Products/Detail.cfm?Prod=NEXYS4
[2] http://www.xilinx.com/products/silicon-devices/fpga/artix-7/

N	L	Accelerator	Octave	Error [%]
100	10	1.351E-030	1.379E-030	1.988
100	20	1.746E-060	1.816E-060	3.834
100	50	3.766E-150	4.146E-150	9.167

Table 6.1. Accuracy of the accelerator using operand widths of $OP_1 = 25$ and $OP_2 = 18$ bits.

N	L	Accelerator	Octave	Error [%]
100	10	1.679E-030	1.682E-030	0.196
100	20	3.043E-060	3.050E-060	0.249
100	50	1.812E-149	1.817E-149	0.286
200	10	6.002E-031	6.005E-031	0.048
200	20	3.624E-061	3.628E-061	0.113

Table 6.2. Accuracy of the accelerator using operand widths of $OP_1 = OP_2 = 25$ bits.

The accuracy is very poor. An error of 10% with values of $N = 100$ and $L = 50$ is not acceptable and needs to be improved. An improvement of accuracy can be achieved by using either a larger operand representation, introducing rounding instead of truncation, or by prescaling operands. As can be seen in in Table 6.1 the error is increasing very fast with a growing L. This points to a problem with the emission probabilities and excludes a truncation error. Operand prescaling could help to reduce the error, but no solution with prescaling has been implemented. This leaves the solution of increasing the operand width. The source code of the implementation was designed in such a way to allow changing the operand width with a minimal effort. In order to limit the increase of resources, only the width of B and TP is increased from 18 to 25 bits. As B is multiplied to the accumulated result at the end of each pipeline stage, it is very probable that the small bit width of B causes this big error. The proposed change doubles the number of Digital Signal Processor (DSP) slices but has no impact on latency and throughput of the accelerator.

Table 6.2 shows the results with an operand width for all operands equal to 25 bit. The accuracy is now in an acceptable range for the chosen values of N and L. With $N = 100$ and $L = 50$ the error is now 0.29%. It would be interesting to see the error for bigger values of L, but unfortunately this is not possible because of the available Hardware (HW).

6.2 Speedup

The results of the accelerator were compared to an Intel CPU of the type i7 3537U. This CPU operates at a frequency of up to 3.1 GHz. The test program was implemented with Octave[3], using matrix operations in order to maximize performance of the serial implementation. The test program was run multiple times in order to allow the CPU to use the cache memory to its full extent.

The accelerator was run at a frequency of 85MHz. Table 6.3 lists in the third column the latency of a likelihood, computed on the accelerator. The next two columns list the delay of

[3] http://www.gnu.org/software/octave/

		Latency [ms]	Delay [ms]			Speedup	
N	L	FPGA	FPGA	CPU (basic)	CPU (scaling)	(basic)	(scaling)
100	10	1.29	0.13	0.28	0.45	2.16	3.48
100	20	2.59	0.13	0.43	0.77	3.32	5.95
100	50	6.47	0.13	0.87	1.71	6.72	13.21
100	100	12.94	0.13	1.5	3.27	11.59	25.27
100	200	25.88	0.13		6.35		48.85
200	10	4.94	0.49	0.42	0.60	0.85	1.21
200	20	9.88	0.49	0.69	1.07	1.40	2.17
200	50	24.71	0.49	1.5	2.46	3.04	4.98
200	100	49.41	0.49	2.9	4.78	5.87	9.67
200	200	98.82	0.49		9.52		19.27

Table 6.3. Speedup of the accelerator using operand widths of $OP_1 = OP_2 = 25$ bits.

consecutive results (the reciprocal value of the throughput) for the accelerator and the CPU. For values of $L > 106$ scaling is required in case of a computation on the CPU. The results with scaling are listed in column six of Table 6.3. The last two columns show the achieved speedup, comparing the accelerator to the basic and the scaled serial implementation respectively. Grey rows represent not real measurements of the computation time of the accelerator but only estimates. The estimation is based on the results computed with lower values of N and L. These results can be scaled by using the knowledge that the computation of one α vector component takes $N^2 + 10N + 2cycles$ to complete. The difference in frequencies of the two devices was not taken into account (only the real measured computation times are considered). In general, the table shows that the accelerator is faster than the serial computation. Using the values for L and N proposed in [24] the estimated speedup is 48.85. Considering the difference in power consumption of the accelerator (estimated consumption of 100mW[4]) compared to the CPU (Thermal Design Power (TPD) of 17W) the results are promising and meet the expectations.

By analysing the table in more detail, big jumps in speedup occur whenever L becomes too big and scaling must also be applied with floating-point numbers. Not only scaling is an issue but also the size of data. The bigger L and N grows, the more likely it becomes that the cache of the CPU is too small to hold all values. At this point, memory access is necessary which increases the computation time considerably.

The throughput of the accelerator depends only on N and is independent of L due to the pipelined architecture. In case of the CPU the throughput decreases with increasing L and N.

To achieve higher speedups the frequency of the accelerator can be increased. This is possible by optimizing the design in order to reduce the critical path. With careful placement of components, introduction of registers or clock buffers, a frequency of up to 250MHz should be possible. To increase the frequency further the automatically generated blocks need to be replaced and designed manually.

6.3 Resources

Table 6.4 shows the required resources in relation with the values of N and L. Grey rows indicate that the values have been estimated while the white rows represent values obtained by

[4] estimated with the ISE design suite of Xilinx

N	L	Slices	Block RAM/FIFO	DSP slices	Example
100	10	2718	27	10	XC7A35T
100	20	5462	47	20	XC7A35T
100	50	13694	107	50	XC7A100T
100	100	27000	207	100	XC7A200T
100	200	54000	407	200	XC7K355T
200	10	2815	65	20	XC7A50T
200	20	5635	85	40	XC7A75T
200	50	14000	145	100	XC7A200T
200	100	28000	245	200	XC7A200T
200	200	56000	445	400	XC7K410T

Table 6.4. Resource requirement of the accelerator using operand widths of $OP_1 = OP_2 = 25$ bits.

performing a synthesis of the Hardware Description Language (HDL) source. The last column of the table provides an example FPGA of appropriate size[5].

[5] http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

Chapter 7

Conclusion

Throughout this thesis a lot of options have been presented, and a lot of decisions have been taken. The vast amount of possibilities in the field of accelerators poses a big challenge to find a solution close to the optimum. One goal of the thesis was to consider multiple viewpoints and trade-offs, and use interdisciplinary knowledge in order to analyse the problems and then take the right choice. A lot of decisions are thoroughly discussed and analysed in the thesis. However, there is still potential to improve and maybe even reconsider some points. The former points will be concluded in Section 7.1 and the latter in Section 7.2.

7.1 Main Contribution

This thesis proposes a novel design of the basic forward algorithm on a Field Programmable Gate Array (FPGA). A fixed-point operand representation and a custom operand scaling scheme is presented. These choices allow to keep the required resources at a low level and limit the energy consumption. The analysis of reasonable parallelization options led to to the choice of a pipelined architecture. The architecture allows to reduce memory access by a factor of L. As the design targets embedded systems, the main focus was to reduce the resource footprint while keeping throughput and latency at an acceptable level. With small values of L and N the accelerator performs slightly better than a General Purpose Processor (GPP). A speedup in the order of 10 can be achieved. For small values of N and L the memory hierarchy of GPPs poses no problem and is able to handle the data flow. The better performance of the accelerator is due to the efficient design and the fixed-point data representation. By increasing N and L the accelerator performs better and reaches higher speedup factors. This increase of speedup is caused by limited availability of fast memory in the memory hierarchy of GPP. Another reason for the increased speedup is scaling. With $L > 106$ (assuming an alphabet size of $M = 1000$) scaling becomes necessary even with double precision representation. For $L = 200$ and $N = 100$, values proposed in [24], a speedup of 48.85 is achieved (this value has only been estimated). Due to the lack of a bigger FPGA test board, the speedup values for bigger L and N have been estimated by scaling the results from the real tests. Scaling the results is a valid approach due to the direct relation of N to the number of required cycles to compute one α vector component. A limiting factor of scalability is on the one hand the available FPGA and on the other hand the critical path. By using a bigger FPGA it becomes more difficult to run the device at high frequencies.

The thesis further proposes a design solution for a failure prediction model that is described in [24]. This includes the design of an extension to the forward algorithm and a memory management system. The extension is designed by adding a computation unit to calculate a new transition probability matrix for every occurring error event. The proposed design of the predictor allows to implement a system capable of predicting one failure type on a FPGA board, by using custom peripherals (Random Access Memory (RAM), Flash, communication interface).

On a conceptional point of view, the thesis provides argumentation on why in a world full of cloud server systems a failure prediction system for a single node can still be beneficial.

7.2 Future Work

The thesis proposes a design and implementation solution for the basic forward algorithm. However, some work is still necessary for being able to use the accelerator as a failure predictor. First and foremost, the extension of the forward algorithm needs to be considered. While a design for the extension of the algorithm is provided in this thesis, an implementation on Hardware (HW) has not been done. Due to the complexity of the extension the design is provided using floating-point blocks. These blocks are not further described in the thesis. References on related work or tools to generate corresponding blocks are provided. To create a consistent design of the extension, a fixed-point design should be performed.

The implementation of the accelerator lacks interfaces to peripherals (e.g. external volatile and non-volatile memories, communication interface) due to the non availability of libraries or necessary licences. In order to use the accelerator in a real case scenario those interfaces must be added. The general approach would be to load the emission probability matrix of size $N \times M$ from a external non-volatile memory into a external volatile memory (e.g. RAM) on power up. The parameters for the Cumulative Distribution Function (CDF) and the initial state probabilities can either be loaded from the external non-volatile memory into the block memory of the FPGA or the data is already available on the FPGA. This can be the case if Read Only Memory (ROM) storage is used or initialized block RAM. Depending on the occurring event, a corresponding emission probability vector is loaded into the FPGA from the volatile memory. The communication interface needs to be set up in order to read occurring error events and write prediction results.

By using a simplified model (cf. 4.7), most of the transition probability matrix entries would be zero. This allows to use a sparse matrix storage technique. A design and implementation of such a technique would reduce the complexity by order $O(N)$. A study of sparse matrix multiplication parallelization methods could prove to be beneficial in order to decrease the latency and the throughput of the accelerator. However, a complete new design would be necessary.

On a more detailed level, some improvements concerning the present implementation could increase performance, lower resource requirements and accuracy:

- By replacing the α vector First In, First Outs (FIFOs) with block RAMs, the design would allow a simple realization of a sparse matrix storage technique. It would also be possible to use only one RAM block as read and write queue for small N.
- Instead of a simple truncation of the operand, a rounding operation would increase the accuracy of the accelerator.

Appendix A

Symbols

- N: number of states of the HSMM
- M: number of observation symbols (size of the alphabet)
- L: observation sequence length
- R: number of CDF (kernels)
- S: number of failure types
- O_k: k-th observation element of the observation sequence $O_1, \ldots, O_L$
- o_j: k-th observation symbol of the set of observation symbols $\boldsymbol{o} = \{o_1, \ldots, o_M\}$
- d_k: delay of the observation element O_k with respect to the observation element O_{k-1}, with $k \in \{1, \ldots, L\}$
- π_i, forming the initial state probability vector $\boldsymbol{\pi}$ 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 $N \times N$
- $d_{ij}(d_k)$, forming the matrix of cumulative transition duration distribution functions $D(d_k)$ of size $N \times N$
- $v_{ij}(d_k)$, forming the extended transition probability matrix V of size $N \times N$
- a_{ij}, forming the basic transition probability matrix A of size $N \times N$
- q_{ij}, forming the transition probability matrix Q of size $N \times N$, addressing both versions (basic and extended)
- $\theta_{ij,r}$: the parameters of the kernel r, with $i, j \in \{1, \ldots, N\}$
- $\omega_{ij,r}$: the weights of the kernel r, with $i, j \in \{1, \ldots, N\}$
- $\kappa_{ij}(d_k|\theta_{ij})$: the kernel, with $i, j \in \{1, \ldots, N\}$
- $\alpha_k(i)$: i-th component of the forward variable vector α_k with $k \in \{1, \ldots, L\}$

- c_k: scaling coefficient of α_k with $k \in 1, \ldots, L$
- λ: the set of parameters of the HSMM
- $P_s(\boldsymbol{o}|\lambda)$: likelihood of the HSMM
- θ_{th}: classification threshold
- $P(c_{\bar{F}})$: 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

Appendix B

Control Signals

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 B.1. A detailed list of all control signals.

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. Vallée, 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] 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.

[21] 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.

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

[23] S. M. ROSS, *Introduction to Probability Models, Tenth Edition*, ch. Estimating Software Reliability.

[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] F. SALFNER, S. TSCHIRPKE, AND M. MALEK, *Comprehensive logfiles for autonomic systems*, in IEEE International Parallel and Distributed Processing Symposium, IPDPS, April 2004, pp. 211–218.

[28] 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.

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

[30] 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.

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

[32] 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.

[33] 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.

[34] Z. ZHENG, Z. LAN, B.-H. PARK, AND A. GEIST, *System log pre-processing to improve failure prediction*, in IEEE/IFIP International Conference on Dependable Systems Networks, DSN, June 2009, pp. 572–577.

Acronyms

ASIC Application Specific Integrated Circuit. 23, 25, 27, 29, 30

BLAS Basic Linear Algebra Subroutines. 6

BV Bit-Vector. 40

CBV Compressed Bit-Vector. 41

CDF Cumulative Distribution Function. ix, xi, 7, 9, 10, 24–27, 30, 34–36, 40, 50, 51

COO Coordinate. 40

CPU Central Processing Unit. 5, 6, 21, 23, 27–30, 45–47

CSR Compressed Sparse Row. 40

CVBV Compressed Variable-Length Bit-Vector. 41

DFT Dispersion Frame Technique. 12

DSP Digital Signal Processor. 23, 28–32, 34, 36, 37, 39, 46, 48

ELL ELLPACK/ITPACK. 40

FIFO First In, First Out. 32, 34, 40, 48, 50

FPGA Field Programmable Gate Array. iii, 5, 6, 23, 26–31, 35, 37, 39, 40, 45, 47–50

GPP General Purpose Processor. 28, 39, 49

GPU Graphics Processing Unit. 5, 6, 21, 23, 26–30

HDL Hardware Description Language. 48

HMM Hidden Markov Model. 5, 7, 9

HPCS High Productivity Computing Systems. 6

HSMM Hidden Semi-Markov Model. iii, 1, 7–9, 30, 51, 52

HW Hardware. ix, 1, 2, 5, 9, 15, 20, 23, 27–31, 35, 41, 42, 46, 50

LUT Look Up Table. 35, 39

LZC Leading Zero Counter. 38, 39

MACC Multiply-Accumulate. 24, 28, 30–32, 37–40

RAM Random Access Memory. 39, 42, 45, 48, 50

ROM Read Only Memory. 50

RTL Register Transfer Level. ix, 31, 33–36, 38, 39

SIMD Single Instruction, Multiple Data Streams. 28

SISD Single Instruction, Single Data Stream. 28

SMO Sequential Minimal Optimization. 6

SVD Singular Value Decomposition. 12

SVM Support Vector Machine. 6, 12

SW Software. 1, 3, 5, 9, 23, 28

TPD Thermal Design Power. 47