Accelerator for Event-based Failure Prediction

Acceleration of an Extended Forward Algorithm for Failure Prediction on FPGA

Master's Thesis submitted to the

Faculty of Informatics of the *Università della Svizzera Italiana*

in partial fulfillment of the requirements for the degree of

Master of Science in Informatics

Embedded Systems Design

presented by

Simon Maurer

under the supervision of

Prof. Miroslaw Malek

Janaury 2014

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

Simon Maurer
Lugano, 29. Janaury 2014

Abstract

Acknowledgements

Contents

Figures

Tables

Chapter 1

Introduction

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

In the event of a system failure it is of course desirable to fix the system as soon as possible in order to minimize the downtime of the system (maximize the availability). This can be accomplished by using different types of recovery techniques, e.g. Check-pointing (create checkpoints to roll back/forward), system replication (switch to a redundant system), fail over (reboot). All these techniques require a certain amount of time to complete the recovery process, time that is very expensive. In order to minimize this time, techniques have been developed to anticipate upcoming failures. Such a technique is described in [20].

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

1.1 Problem Statement

1.2 Motivation

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

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

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

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

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

1.3 Contributions

1.4 Document Structure

Chapter 2

State of the Art

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

2.1 Failure Prediction

A very detailed overview of failure prediction methods is given in [21]. The survey discusses i.a. the techniques used as comparison in the main reference [12, 11, 23, 6] as well as the technique described in the main reference [20].

More recent work uses hardware counters of a general purpose CPU and combines them with software instrumentation to analyze failures of single processes (e.g grep, flex, sed) [26]. As industry heads more and more towards cloud computing, it has been proposed to use information of interaction between nodes (instead of analyzing single nodes) in order to analyze and predict failures of a distributed system [22, 16].

2.2 Accelerator

The main goal of this master thesis is to accelerate an adaptation of the forward algorithm. Proposals for a GPU based accelerator for the classic forward algorithm are described in [15, 13]. 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, [7, 14, 17] describe a FPGA based accelerator for protein sequence HHM search and [24] describes i.a. an approach to accelerate the Viterbi algorithm from the HMMER library using GPUs.

Focusing on a more general approach for acceleration, [9] proposes an FPGA implementation of a parallel floating point accumulation and [25] describes the implementation of a vector processor on FPGA.

Quite some research has been done on the question what type of technology should be used to accelerate certain algorithms: [4] presents a performance study of different applications accelerated on a multicore CPU, on a GPU and on a FPGA, [8] discusses the suitability of FPGA and GPU acceleration for high productivity computing systems (HPCS) without focusing on a

specific application and [10] also focuses on HPCS but uses the Basic Linear Algebra Subroutines (BLAS) as comparison and also takes CPUs into account.

It may be interesting to also think about an acceleration of the model training. Similar work has been done by accelerating SVMs (Support Vector Machines): [3] describes a FPGA based accelerator for the SVM-SMO (support vector machine - sequential minimal optimization) algorithm used in the domain of machine learning and [1] proposes a new algorithm and its implementation on a FPGA for SVMs.

Chapter 3

Event-based Failure Prediction

This section provides a brief overview of the computational steps done by the proposed algorithm [20].

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

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

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

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

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

3.1 Data Processing

3.2 Model Training

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

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

3.3 Sequence Processing

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

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

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

where

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

with

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

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

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

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

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

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

maybe use sequence likelihood and then explain about scaling

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

$$\alpha_k(i) = c_k \alpha_k(i) \tag{3.8}$$

with

$$c_k = \frac{1}{\sum_{i=1}^{N} \alpha_k(i)} \tag{3.9}$$

begin new sentence and explain log likelihood

then the sequence log-likelihood is computed:

$$\log(P(\boldsymbol{o}|\lambda)) = -\sum_{k=1}^{L} \log c_k \tag{3.10}$$

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

3.4 Classification

explain classification

and finally the classification is performed:

$$\text{class}(s) = F \iff \max_{i=1}^{u} \left[\log P(\boldsymbol{s}|\lambda_i) \right] - \log P(\boldsymbol{s}|\lambda_0) > \log \theta \tag{3.11}$$

with

$$\theta = \frac{(r_{\bar{F}F} - r_{\bar{F}\bar{F}})P(c_{\bar{F}})}{(r_{F\bar{F}} - r_{FF})P(c_F)} \tag{3.12}$$

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

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

Chapter 4

Acceleration

Challanges of the acceleration

- implementation of exp and log function (LUT, Taylor, ...)
- floating points vs fixed points
- precision
- choice of accelerator (Type, Model)
- find avaliable options for parallelization

Ideas on how to accelerate the online part of the algorithm

- use high speed multiplier-accumulator (MAC) devices on a FPGA
- use MACs only on integer numbers and compute FP later manually
- minimize division (compute scaling factor once and then multiply)
- if precision allows use pipelining to precompute the factor $b(j, o[k]) * v(i, j, k)$
- precompute known factors and store them in order to simlify online computation (e.g. parts of the kernel, classification thershold, ...).
- ...

Possible optimizations of the algorithm:

- use a regularization term in the cost function to prevent overfitting
- incorporate the offline part of the algorithm into the online part in order to deal with model aging
- ...

4.1 Theoretical Analysis

4.1.1 Serial Implementation

Forward Algorithm

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% computation of the extended forward algorithm
%
% @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 cdf_param:     parameters for the cdf
% @return alpha:        forward variables. size N, L
% @return scale_coeff:  scaling coefficients (needed for log likelihood). size L
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [alpha lPs] = forward_s(N, L, PI, B, dL, oL, cdf_param)
    k = 1;
    % read o0: index of first observation symbol
    % initialize forward variables
    [alpha(:, 1) scale_coeff(1)] = forward_s_init(N, PI, B(:, oL(1)));
    % forward algorithm
    while (k < L),
        % read ok: index of k-th observation symbol
        % read dk: delay of k-th observation symbol
        % compute one step of forward algorithm
        [alpha(:, k+1) scale_coeff(k+1)] = ...
            forward_s_step(N, dL(k), alpha(:, k), B(:, oL(k)), cdf_param);
        k++;
    end
    % compute log likelihood
    lPs = 0;
    for i=1:N,
        lPs -= log(scale_coeff(i));
    end
end
```

Initialisation of each step

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% computation of the initial values of the forward algorithm
%
% @param N:             number of states
% @param PI:            initial state probability vector. size N
% @param B:             matrix of emission probabilities of step 0. size N
% @return alpha:        initial forward variables. size N
% @return scale_coeff:  initial scaling coefficient
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [alpha scale_coeff] = forward_s_init(N, PI, B)
    for i=1:N,
        alpha(i) = PI(i)*B(i);
    end
```

```
    % scaling
    alpha_sum = 0;
    for i=1:N,
        alpha_sum += alpha(i);
    end
    scale_coeff = 1 / alpha_sum;
    for i=1:N,
        alpha(i) *= scale_coeff;
    end
end
```

Computation per Observation Symbol

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% computation of one step of the extended forward algorithm
%
% @param N:             number of states
% @param dk:            delay of k-th observation symbol
% @param alpha:         forward variables of step k-1. size N
% @param B:             emission probabilities of step k. size N
% @param cdf_param:     parameters for the cdf
% @param alpha_new:     forward variables of step k. size N
% @return scale_coeff:  scaling coefficient of step k
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [alpha_new scale_coeff] = forward_s_step.m(N, dk, alpha, B, cdf_param)
    % compute transistion probabilities
    tp = compute_tp(N, dk, cdf_param);
    % compute forward algorithm
    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);
    end
    % scaling
    alpha_sum = 0;
    for i=1:N,
        alpha_sum += alpha_new(i);
    end
    scale_coeff = 1 / alpha_sum;
    for i=1:N,
        alpha_new(i) *= scale_coeff;
    end
end
```

Extension of Forward Algorithm

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% computation of the extended transition probabilities
%
% @param N:         number of states
% @param dk:        delay of k-th observation symbol
```

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

Script to run a sliding window over the sequenze

```
N = 100;
L = 100;
PI = read_PI();
B = read_B();
cdf_param = read_cdf();

while(symb = read_next())
    dL = [dL(2:end); symb.d];
    oL = [oL(2:end); symb.o];
    lPs = forward_s(N, L, PI, B, dL, oL, cdf_param);
end
```

4.1.2 Avaliable Parallelism

4.1.3 Scaling

Division is expensive but needed if scaling is applied. Scaling is useful to prevent that the continuous multiplication of floating points results in zero.

If scaling is used, compute a scaling factor (only one division) and the use multiplication to scale (because division is a lot more complex than the multiplication).

Keep the precision of the operands as low as possible to save memory and to reduce computation complexity.

4.1.4 Computation of Extension

This computation is very expensive but needs only to be computed once per dk. (Unlike the alphas, which need to be recomputed for the same dk because they depend on the previous

alpha).

Maybe use a very spezialized unit (ASIC) just for the computation of the cumulative distribution function.

4.2 Choice of Accelerator Type

4.2.1 CPU

- pro
  - fast and easy implementation
  - high precision
  - high frequency
- contra
  - high power consumption
  - limited parallelization
  - large overhead for simple instructions
  - fixed architecture (memory, computation units)

4.2.2 GPU

- pro
  - parallelization options for a low price
  - fast onboard memory
  - high frequency
  - high precision
  - simple implementation
- contra
  - high power consumption
  - overhead for simple instructions
  - fixed architecture (memory, computation units)

4.2.3 FPGA

- pro
  - low power consumption
  - low overhead
  - flexibility
  - optimized floating point representation (small values)

- contra
  - low frequency
  - parallelization is expensive
  - precision is expensive
  - complex implementation

4.2.4 ASIC

- pro
  - very low power consumption
  - no overhead
  - very flexible
  - optimized floating point representation (small values)
- contra
  - very expensive
  - very complex implementation

4.2.5 Conclusion

4.3 Implementation

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

4.3.1 Design

- parallelization of one of nested for-loop or pipeline the loops
- fully pipelined MACC

The computation of the likelihood is divided in L steps: the initialization, $L-2$ identical intermediate steps and the finalization. N initial forward variables α_0. Each intermediate step computes N intermediate forward variables α_k and the final step calculates the last set of N forward variables α_L as well as the likelihood. Each step takes the emission probabilities $b_i(o_k)$ corresponding to the observation symbol o_k and constant factors as input. 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, as at every arrival of a new observation symbol, the last L elements of the observation symbol sequence are used to compute the likelihood, it is very beneficial to pipeline the steps.

To do this, each step is realized in hardware and connected as represented in the figure 4.1. With this configuration, a likelihood is computed at every completion of a step with a latency of $L * t_{step}$, where t_{step} is the time needed to complete the computation of one step. Additionally the configuration allows to load the constants TP and the emission probabilities $b_i(o_k)$ for all steps at the same time, which reduces the load operations considerably.

example on how the pipeline works

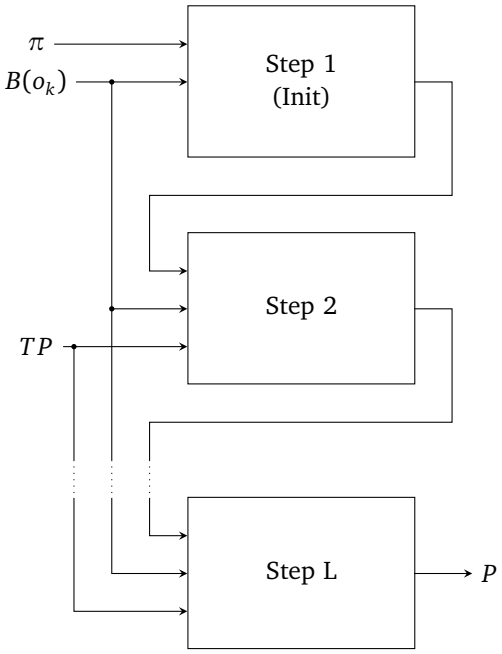

Figure 4.1. Top View of the Architecture

Following, each dissimilar block is described in more detail, beginning with the initialization step. Figure 4.2 shows the necessary operation every i-th component.

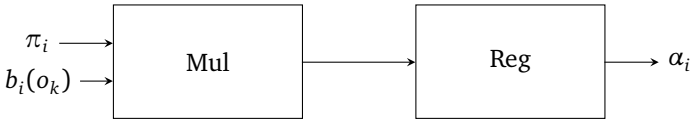

Figure 4.2. Initialization step

4.3.2 Precision

- use non-standard floating point representation, unsigned
- increase exponent size to represent very small numbers
- don't use scaling (division is expensive)
- mantissa of 1st multiplication operand: 25bit

- mantissa of 2nd multiplication operand: 18bit
- exponent of stored values: 8bit, unigned
- exponent of computed values: tbd, unsigned

4.3.3 Data Storage Management

- real-time (time constraints on every Ps) vs on-line (use buffer to optimize throughput)
- memory heirarchy: at startup copy from flash to ram, then use pipeline to preload values from ram into registers.

Chapter 5

Testing and Verification

5.1 Log Standard

5.2 Metrics

5.3 Automated Log Generation

5.4 Online Log Generation

Chapter 6

Results

6.1 Speedup

6.2 Accuracy

Chapter 7

Conclusion

7.1 Achievements

7.2 Future Work

Appendix A

Some material

Bibliography

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] B. OZCELIK AND C. YILMAZ, *Seer: A lightweight online failure prediction approach*. http://research.sabanciuniv.edu/23359/, January 2014.

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

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

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

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

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

[24] 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.

[25] 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.

[26] 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.