Using artificial neural networks for error detection in unreliable computations

Vassilis Vassiliadis, Konstantinos Parasyris, Christos D. Antonopoulos, Spyros Lalis, Nikolaos Bellas
University of Thessaly,
Volos, Greece
{*vasiliad, koparasy, cda, lalis, nbellas*}*@e-ce.uth.gr*

***Abstract*—We introduce a methodology to use Artificial Neural Networks (ANNs) for automatic error detection on outputs of selected parts of a program which execute on unreliable hardware that operates at frequencies beyond the nominal levels. We use an OpenMP inspired programming model and an accompanying runtime system which enables developers to specify the significance of tasks regarding their effect to the output quality of the program. The runtime system executes the least significant tasks on cores which operate unreliably and uses specially trained ANNs to detect errors which are then corrected by a user supplied error correction function. We test our approach using with a benchmark application that uses the Discrete Cosine Transform (DCT) kernel to compress images. A fault injection campaign indicates that it is possible to achieve a 1.77x speedup over a fully reliable execution of the code, with a minimal penalty to output quality (43.46 dB Peak Signal to Noise Ratio between the de-compressed and the original image compared to the 43.67 dB PSNR of a fully reliable execution).**

***Keywords*-Fault tolerance, Software reliability, Software performance, Artificial neural networks**

I. Introduction

Traditionally computer systems are expected to function reliably. However, as datacenters grow rapidly and are expected to grow by 50x in the next years, accomplishing such strict reliability requirements becomes harder. Recent studies suggest that if an Exascale system was created using today's components, its time to failure would be between 3 and 37 minutes [16], [2]. Corrupted data are also expected to appear more frequently [3].

In High End Computing with parallel computations on clusters with 10s/100s of thousands of cores, errors have become a norm and not the exception [6]. Prior work demonstrates that the execution time spend to perform checkpointing/restoring procedures ranges from 85% to 55% of the total execution time of the application [4].

Consequently, there is a need to create an efficient methodology to detect errors during execution time. This work uses Artificial Neural Networks (ANNs) to detect errors during the execution of code. We propose a methodology to train ANNs whose goal is to detect errors that negatively effect the output of unreliably executed code. Then we apply our methodology on a multimedia benchmark which uses Discrete Cosine Transform and quantization to compress images (**DCT**). We evaluate our approach using software fault injection, to simulate an unreliable environment. Finally, we measure the benchmark speedup and output quality degradation.

Our goal is to increase application performance by setting the processor to higher frequency without increasing its voltage supply. However, since the hardware operates at a much higher frequency than it can reliably support, hardware errors might occur.

Using a software fault injection approach, we measure the speedup obtained via Unreliable Computing coupled with Artificial Neural Networks to be on average 1.82x compared to a fully Reliable execution. Images obtained through our software hardening methodology have high output quality with an average Peak Signal to Noise Ratio (PSNR) value of about 43.46 dB. A respective error-free execution produces images with 43.67 dB PSNR.

The rest of the paper is structured as follows. Section II discusses Unreliable Computing. Section III details our approach to automatic error detection via ANNs. In Section IV we evaluate our methodology using **DCT** as an example benchmark. Afterwards, in Section V we provide an overview of related work. Finally, Section VI concludes this work and presents directions for future research.

II. Unreliable Computing

The performance increase of modern hardware is mostly due to the effects of reducing the size of semiconductors. However, scaling down the semiconductors increases the manufacturing variability which results in less deterministic transistor characteristics. As a result, chip yield is reduced, and manufacturers have to counter these issues with increased voltage margins and guard bands. Such guard bands typically account for the worst case scenario.

Our approach to Unreliable Computing exploits the Significance [17] of operations as an algorithmic property to gracefully trade-off output quality with execution time reduction. Significance of operations is a quantitative metric for measuring the effect that a code has to the output quality. A highly significant code produces data which if slightly perturbed lead to a large change on the output quality.

In the general case, executing computations on unreliable hardware is unpredictable. There is a chance that a code may terminate successfully and produce 100% correct results. However, it may also terminate abruptly and fail to produce any output. Another possible scenario is that

the code enters an infinite loop and never terminates, or even produces inaccurate results which in the literature are called Silent Data Corruptions (SDCs). A requirement to using Unreliable Computing efficiently is to gracefully trade-off output quality with power/energy efficiency. We achieve this requirement by only executing code using unreliable hardware when it does not heavily influence the final output quality. In other words, only the least significant parts of an application may be executed unreliably to minimize the exposure of significant computations to errors.

We adopt and expand on the task-based programming paradigm of OpenMP [12]. Our programming model enables developers to explicitly declare the significance of computations at the granularity of tasks, depending on how strongly they contribute to the quality of the end result. The programming model supports error-checking and correction mechanisms to mitigate the effects of executing tasks on unreliable hardware to the final output quality. An accompanying runtime system executes less significant tasks on unreliable but faster cores and the rest of the code on reliable hardware. Error detection and correction can be realized using result-check and correct functions which are invoked by the runtime system on task completion or failure.

A. Error detection for unreliable tasks

Simply executing only the least significant parts of an application using unreliable hardware is not enough to guarantee that the program produces acceptable output. An error, even if it manifests during the execution of a non-significant part of the application may still severely hinder the quality of the output. It follows that, hardware faults have notoriously unpredictable effects. To this end, applications need also detect and correct errors that happen during the execution of their unreliable portions. This introduces an overhead to the execution of the application, which is conceptually similar to the guard-bands and over-provisions that hardware designers introduce to their work, albeit at the software level.

Recent studies have focused on designing mechanisms that allow the execution of code on top of unreliable hardware while gracefully handling errors as well as detection and correction of SDCs [15], [9], [1], [11], [14].

In this work we investigate the efficiency of ANN assisted error checking. ANNs, in their purest form, can be seen as a sets of nodes in pipeline. Each set, or stage, in the pipeline is a Neural Network Layer. ANNs were designed to loosely model the functionality of the network of neurons in a brain. This biologically inspired tool can theoretically approximate any function by observing its inputs and outputs and adjusting its nodes' weights, achieved during a process that is called training. In this work, we propose and test the use of ANNs which we first train to detect errors to the output of tasks.

Algorithm 1: Error Detection using Artificial Neural Networks

Input : Application source code, Src
Output: Hardened application source code

S1: Src_{tasks} = partitionCodeIntoTasks(Src)
S2: $outputs$ = performSoftwareInjection(Src_{tasks})
S3: $outputs_{labeled}$ = label($outputs$)
S4: $outputs_{normalized}$ = normalize($outputs_{labeled}$)
S5: $ANNS$ = generate()
S6: $ANNS_{trained}$ = train($ANNS$, $outputs_{normalized}$)
S7: $Functions$ = convertANNtoC($ANNS$)
S8 Src_{batch} = groupTasksInBatches(Src_{task})
S9: Src_{hardnd} = attachANNFnc(Src_{batch}, $Functions$)

B. Error correction

When a task's output are detected to be erroneous, corrective action must be taken so that the faulty values do not propagate to subsequent computations so that no output quality degradation occurs.

The simplest method of correcting a task that was executed unreliably is simply to re-execute it on reliable hardware. However, this negates the benefits of executing on top of unreliable hardware. Such a correcting step should only be taken when tasks are rarely found to be faulty. On the other hand, if the chosen error detection mechanism is inaccurate there is the possibility of failing to re-execute tasks which have produced wrong outputs. As such, even if task re-execution is used to correct errors the output of partially unreliable execution of an application might still differ than a fully reliable one.

An alternative to re-executing tasks is to exploit programmer wisdom in order to create an approximate alternative of the original task code which is less accurate but cheaper in terms of computational costs. Such approximations might also be generated using automatic software [17].

III. Methodology

The ideal error detection function identifies all errors and never mis-characterizes correct output values as erroneous. In the general case, designing and implementing such a function is easier said than done. Manually implementing error detection and correction functions for an application is a time consuming and tedious process.

Our proposed methodology exploits the fact that an Artificial Neural Network (ANN) may, potentially, approximate the solution to any problem given enough observational data and computational complexity in the form of a complex artificial neural network topology. Algorithm 1 shows the pseudo-algorithm that drives our training methodology for ANNs.

We start by partitioning the original code into tasks (**S1**). Each task must have well defined inputs and outputs whose size does not change for different program inputs. We use Caffe [8] to train and generate ANNs which detect errors on the outputs of tasks. These ANNs consist of a single input layer, one or more hidden (intermediate) layers, and a single output layer. The hidden layers are either Fully Connected layers or Rectified Linear Unit (ReLU) layers. Note that, the requirement of having fixed size task inputs and outputs can be eliminated by employing Recurrent Neural Networks [7]. In this work, we do not explore this type of ANN because it is an orthogonal choice to error detection.

Steps **S2** through **S4** generate the training and test data of the ANNs. In **S2** the application undergoes a software fault injection campaign whose purpose is to generate enough observational data to be used during the training phase.

Step **S3** uses an error-free execution of the application to label the data produced in step **S2** as acceptable, or unacceptable in a per-task fashion. Acceptable results differ slightly from the corresponding error-free output. In this document we used 2% as the selected threshold; we plan to explore different trade-ofs by varying this threshold in future work. Allowing small deviation from the correct value enhances the generality of the ANN and reduces the chances of over-fitting the ANNs. An over-fitted network performs great when it operates on its trained data but its efficiency is greatly diminished when it encounters unseen data.

S4 performs data normalization by means of scaling the labeled data produced in step **S3** to the $[-1, 1]$ range. Since the data are computed under the presence of errors we only consider error-free entries to calculate the maximum and minimum values. We also make sure that the training data set is equally distributed between acceptable and non-acceptable entries. It is nearly impossible to have a tight bound for all possible, including potentially erroneous, data. Consequently, at runtime during the scaling of ANN inputs we forcefully clip values to the range $[-10, 10]$ to keep the inputs of the ANN used within normal value-range.

Step **S5** generates a set of ANNs. Our algorithm generates hidden layers in pairs, we refer to such a pair as an Artificial Neural Network Level (ANNL). Each ANNL consists of a Fully Connected (FC) layer which is directly followed by a ReLU layer. The size of the ANNL is the number of nodes that form each layer, e.g. an ANNL of size 4 contains a FC layer of 4 nodes followed by a 4 node ReLU layer. To limit the exploration space of possible topologies we generate ANNs which contain from zero up to two ANNLs. The size of each ANNL is either a power of two between the range $[4, 32]$ or 0. Thus, when targeting at most 2 ANNLs our methodology generates 21 ANN configurations in total plus one FC layer right after the input layer, and one FC layer to produce the output layer.

At this point the ANNs generated by the last step of the process undergo training in step **S6**. We use Caffe to perform forward and back-propagation of the ANNs on the train data-set. Every 1000 train epochs we perform the forward operation of the ANNs using the test data as input and record their accuracy. In this context, the accuracy of the ANN is defined as the relative error as computed between the ANN's output for a given input compared to the correct output classification. We keep track of the best obtained accuracy and a snapshot of the ANN at the best epoch. The process ends when no improvement to the ANN's accuracy is recorded within 25000 epochs.

In step **S7** the framework generates a function (in C) that performs the computations of an ANN and reports whether a task output is acceptable or not. This effectively removes any run-time dependencies to Caffe and enables the inspection and analysis of the error detection function. Furthermore, the code which performs the operations of the ANN is fully unrolled. It comprises multiplication and addition operations between floating point data for each FC layer and binary instructions for each ReLU layer. This enables the use of fixed point arithmetic to further optimize the execution time of the error detection phase as well as the use of specialized hardware for reduced energy/power cost.

The last two steps of the process (**S8** and **S9**) involve structural changes to the application code as produced by **S1**. In **S8** since tasks might be too fine grained for practical use the application developer is prompted to group the tasks into task-batches. In **S9** the final versions of the application are finished by hardening them against errors through the C error detection functions generated in step **S7**

At this point, the application developer has to decide what happens when an error is detected. He can either re-execute tasks which are considered erroneous, or try to correct the errors by means of approximating the faulty task.

The results of fault injection campaign are analyzed to choose the ANN which results in the desired speedup and output quality level. We discuss more on this in the following section which uses **DCT** as a running example.

IV. Experimental Evaluation

We evaluate ANN assisted error detection using Discrete Cosine Transform (**DCT**) as a running example. **DCT** is a module of video compression kernels, which transforms a block of image pixels to a block of frequency coefficients. The tasks that compute low frequency coefficients, close to the upper left corner of each 8x8 frequency block, are more significant than the ones computing coefficients towards the lower right corner of the block because the human eye is more sensitive to low frequency changes. We instruct the runtime to execute tasks which compute the top-left corner of each 8x8 frequency block reliably, and all remaining tasks unreliably. Unreliable tasks that are declared erroneous by the ANNs have their output set to zero. Finally, infinite loops are handled using a timeout functionality of our programming model at task synchronization points.

We simulate an unreliable hardware environment through the use of software fault injection. For the hardware community, the Point of First Failure (PoFF) indicates the point at which circuits start to exhibit massive errors. More specifically, one error every $\sim$10 million cycles [5]. Prior to this point errors still occur, however at rates that are orders of magnitude lower. The PoFF is met when the nominal supply voltage differs from the actual supply voltage by 15%. Based on these findings we use an Intel(R) Core(TM) i7-4820K CPU @ 3.70GHz processor to perform software fault injection. We select two nominal points, one is used as the reliable but slow domain and the other as the unreliable but fast domain. Their nominal supply voltage differs by 15%, the slow and reliable domain being ($frequency_{low} = 1.6GHz, voltage_{low} = 0.9V$) and the fast but unreliable domain ($frequency_{high} = 3.7GHz, voltage_{low} = 0.9V$). This effectively emulates a processor whose highest frequency that can be executed reliably is 1.6GHz but can be scaled to 3.7GHz for an unreliable execution. Using this information we perform a fault injection campaign and measure the number of cycles spent in each domain to synthetically compute the execution time of experiments.

For **DCT** we always execute the most significant tasks of **DCT** reliably which amounts to exactly 12.5% percent of total tasks. To determine the fault injection campaign's number of experiments we consult [10] and opt for a 95% confidence level and 2.5% error margin which amounts to a total of 31815 experiments. Upon the completion of the fault injection campaign, we collect the execution time and output quality metrics and then select the top 8 ANN topologies with respect to the output quality.

In Figure 1 we show the probability of **DCT** producing output images with at least a specific Peak Signal to Noise Ratio (PSNR) for the 4 most interesting ANN topologies. An error free execution of the benchmark yields an output quality of 43.67 dB (Figure 3). With this in mind we can see that 2 out of the 21 ANNs, consisting of the layer topologies $8, 32, 8, 1$ and $8, 32, 16, 1$, result in PSNR values that are higher than 43.6 dB with a 99.93% probability. But these are the two slowest ANNs, as seen in Figure 2. Notice that the ANN with layers $8, 4, 4, 1$, produces images with a 43.46 dB PSNR (Figure 4) but at a much higher speedup of 1.82X. Consequently, it makes sense to use this ANN for error-detection as it computes almost indistinguishable outputs compared to error-free executions of the application with a much smaller execution time. In Figure 5 we show the differences between Figure 3 and Figure 4. Notice that errors appear when there are blocks of pixels with high frequency changes, like the separation between sky and sea level and rough textured rocks.

Another interesting point, which is illustrated in Figure 2, is that worse results do not necessarily come with the benefit of faster execution as is typically the case with Approximate Computing. Even though the $8, 8, 8, 1$ ANN is slower than $8, 4, 4, 1$ it does not lead to better output quality levels.

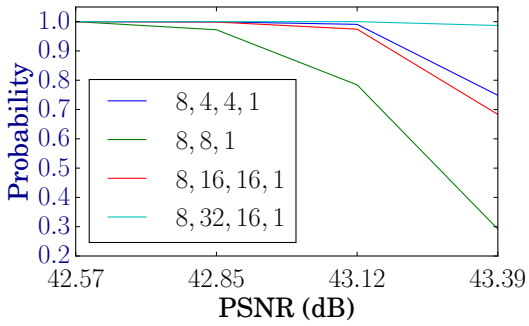

Figure 1: The probability that an ANN results in images with at least a specific PSNR value. The first (8) and last layer (1) of each ANN corresponds to the input and output layers respectively. The ANNs are listed in decreasing execution time see Figure 2 for a more detailed speedup, output quality comparison

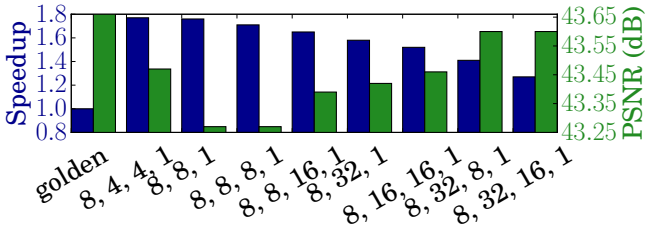

Figure 2: Speedup vs average quality comparison for the top 8 ANN topologies in terms of output quality. The X axis indicates the ANN topologies. The left Y axis shows the speedup with respect to a golden/error-free execution. The right Y axis shows the average output PSNR

V. Related work

Two offline debugging mechanisms and three online monitoring mechanisms for approximate programs are presented in [15]. Among the offline mechanisms, the first one identifies correlation between QoR and each approximate operation by tracking the execution and error frequencies of different code regions over multiple program executions with varying QoR values. The second mechanism tracks which approximate operations affect any approximate variable and memory location. The online mechanisms complement the offline ones and they detect and compensate for QoR loss while maintaining the energy gains of approximation. The first mechanism compares the QoR for precise and approximate variants of the program for a random subset of executions. This mechanism is useful for programs where QoR can be assessed by sampling a few outputs, but not for those that require bounding the worst-case errors. The second mechanism uses programmer-supplied "verification functions" that can check a result with much lower overhead than computing the result. The third mechanism stores past inputs and outputs of the checked code and estimates the

Figure 3: Output of error-free **DCT** (43.67 dB PSNR)

Figure 4: Output of partially unreliable **DCT** using the $8, 4, 4, 1$ ANN to detect errors (43.46 dB PSNR)

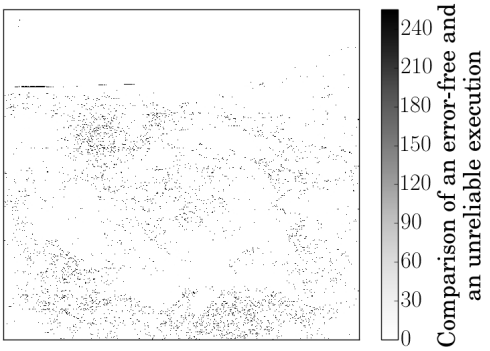

Figure 5: Heatmap of the difference between Figures 3 and 4

output for current execution based on interpolation of the previous executions with similar inputs. They show that their offline mechanisms help in effectively identifying the root of a quality issue instead of merely confirming the existence of an issue and the online mechanisms help in controlling QoR while maintaining high energy gains. Our method could also be applied to detect errors due to approximation but in this work we focus on errors induced by unreliable execution.

[9] presents an output-quality monitoring and management technique which can ensure meeting a given output quality. Based on the observation that simple prediction approaches, e.g. linear estimation, moving average, and decision trees can accurately predict approximation errors, they use a low-overhead error detection module which tracks predicted errors to find the elements which need correction. Using this information, the recovery module, which runs in parallel to the detection module, re-executes the iterations that lead to high-errors. This becomes possible since the approximable functions or codes are generally those that simply read inputs and produce outputs without modifying any other state, such as map and stencil patterns. Our approach differs in that we use an ANN to detect error whereas [9] uses hardware accelerated ANNs to approximate code whose output is subsequently error checked. Large errors on the approximated computations are corrected by means of executing the accurate code using the CPU.

Topaz [1] is a task-based framework which executes unreliably computations. An online outlier detection mechanism detects and corrects, through re-execution on reliable hardware, unacceptable task results. Chisel [11] is a framework which given a reliability and/or accuracy hardware specification automatically selects approximate kernel operations to synthesize a computation which minimizes energy consumption and satisfies the accuracy specification. Our framework dynamically operates at different energy gain/output qualities configurations by selecting at runtime a different reliable/unreliable task mix to gracefully trade-off output quality with energy reduction.

Rinard, in one of the chronologically earlier efforts on task-based error-tolerant computing, proposes a software mechanism that allows the programmer to identify task blocks and then creates a profile-driven probabilistic fault model for each task [14]. This is accomplished by injecting faults at task execution and observing the resulting output distortion and output failure rates. The concept of Task Level Vulnerability (TLV) captures dynamic circuit-level variability for each OpenMP task running in a specific processing core [13]. TLV meta-data are gathered during execution by circuit sensors and error detection units to provide characterization at the context of an OpenMP task. Based on TLV meta-data, the OpenMP runtime apportions tasks to cores aiming at minimizing the number of instructions that incur errors. Although, similar to our approach, this work does not consider error recovery and user-specified approximate execution paths.

VI. Conclusions

We have presented preliminary results which indicate that it is possible to automate the process of error checking

unreliable computations through the use of ANNs to gracefully trade-off performance with output quality. The best performing ANN lead to a minuscule decrease in PSNR of about 0.21dB while allowing for a 1.77x speedup over a fully reliable execution of the application.

It is our immediate plan to investigate how our methodology fairs with a larger set of benchmarks. Beyond that, it would be interesting to devise an intelligent metric which combines the cost of operating ANNs and the output quality that they achieve so that the selection of the best ANN becomes more formal.

REFERENCES

[1] S. Achour and M. C. Rinard. Approximate computation with outlier detection in topaz. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 711–730, New York, NY, USA, 2015. ACM.

[2] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, et al. Exascale software study: Software challenges in extreme scale systems. *DARPA IPTO, Air Force Research Labs, Tech. Rep*, 2009.

[3] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 2009.

[4] J. T. Daly, L. A. Pritchett-Sheats, and S. E. Michalak. Application mttfe vs. platform mtbf: A fresh perspective on system reliability and application throughput for computations at scale. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*, pages 795–800, May 2008.

[5] S. Das, D. Roberts, S. Lee, S. Pant, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. A self-tuning dvs processor using delay-error detection and correction. *Solid-State Circuits, IEEE Journal of*, 41(4):792–804, 2006.

[6] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[7] L. C. Jain and L. R. Medsker. *Recurrent Neural Networks: Design and Applications*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1999.

[8] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[9] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. Rumba: An online quality management system for approximate computing. *SIGARCH Comput. Archit. News*, 43(3):554–566, 2015.

[10] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: quantified error and confidence. In *DATE '09.*, pages 502–506. IEEE, IEEE, Apr 2009.

[11] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. *SIGPLAN Not.*, 49(10):309–328, Oct. 2014.

[12] OpenMP Architecture Review Board. OpenMP Application Program Interface (version 4.0). Technical report, July 2013.

[13] A. Rahimi, A. Marongiu, P. Burgio, R. K. Gupta, and L. Benini. Variation-tolerant OpenMP Tasking on Tightly-coupled Processor Clusters. In *DATE '13*, pages 541–546. EDA Consortium, 2013.

[14] M. Rinard. Probabilistic Accuracy Bounds for Fault-tolerant Computations That Discard Tasks. In *ICS '06*, pages 324–334. ACM, 2006.

[15] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman. Monitoring and debugging the quality of results in approximate programs. In *ASPLOS '15*, pages 399–411. ACM, 2015.

[16] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In J. Palma, M. Dayd, O. Marques, and J. Lopes, editors, *High Performance Computing for Computational Science VECPAR 2010*, volume 6449 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin Heidelberg, 2011.

[17] V. Vassiliadis, J. Riehme, J. Deussen, K. Parasyris, C. D. Antonopoulos, N. Bellas, S. Lalis, and U. Naumann. Towards automatic significance analysis for approximate computing. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 182–193, New York, NY, USA, 2016. ACM.