2019 IEEE 37th VLSI Test Symposium (VTS)

# Special Session: Does Approximation Make Testing Harder (or Easier)?

R. Iris Bahar
*School of Engineering*
*Brown University*
Providence, RI
iris_bahar@brown.edu

Ulya Karpuzcu
*Dept. of Elect. and Comp. Engineering*
*University of Minnesota Twin Cities*
Minneapolis, MN
ukarpuzc@umn.edu

Sasa Misailovic
*Dept. of Computer Science*
*University of Illinois*
Urbana, IL
misailo@illinois.edu

*Abstract*— **Many important application domains, including machine learning, feature intrinsically noise tolerant algorithms. These algorithms process massive, yet noisy and redundant data, by probabilistic and often iterative techniques. As a result, there is a range of valid outputs rather than a single golden value. While this may translate into relaxed constraints for testing and verification of approximate systems, distinguishing actual design bugs from what is being approximated also becomes harder. In this paper, using representative case studies, we pose several challenges for the test and verification community as approximate computing becomes more prevalent as a design of choice in order to achieve performance gains, power or energy savings, improved reliability or reduced software and/or hardware complexity.**

## I. An Overview Across the System Stack

Intrinsic algorithmic noise tolerance is a common characteristic in the vast majority of emerging application domains, including but not necessarily limited to machine learning. Such algorithms often feature probabilistic techniques in processing their noisy (and often redundant) input data. Therefore, there exists a range of valid output values rather than a single golden value. Usually this range of valid outputs is bounded by an $\pm$ *acceptable* deviation from (what would otherwise be) the golden value, where acceptability itself is application-dependent. This deviation directly translates into loss in computational accuracy.

In the following, we will use *approximate computing* to cover a diverse set of techniques across the system stack, which trade computational accuracy for performance gains, power or energy savings, improved reliability or reduced software and/or hardware complexity. Promising representatives include precision reduction [21], [49], [61], [62]; computation perforation [6], [15], [48], [51]; relaxation of execution semantics [35], [46]–[48] often accompanied by hardware simplification [5], [11], [17]–[19], [21]; and embrace of errors [1], [3], [4], [29], [30], [33], [37], [42]. For further discussion and broader coverage, we refer the reader to many surveys on the subject matter such as [7], [40], [52], [59].

Quantification of approximation-induced accuracy loss using application-specific metrics is already a very critical step in harvesting the intrinsic noise tolerance for better overall system efficiency. This inevitably necessitates the development and deployment of accuracy metrics. Almost all approximation techniques use accuracy metrics to justify the approximations and show their benefit by studying the accuracy-performance tradeoffs.

Accuracy metrics are equally critical when it comes to functional testing and verification. In contrast to most software, for which the developers can test for a binary notion of correctness (e.g., by ensuring that the results are the same as the expected results), approximate computations can produce a range of *acceptable* results. Accuracy metrics check whether the result is within the acceptable range or whether the frequency of errors is acceptably small. On one hand, validation against a range of values (as opposed to a golden value) using accuracy metrics may ease testing and verification of approximate systems by relaxing constraints. On the other hand, distinguishing actual design bugs from what is being approximated becomes harder, as design bugs may result in very similar deviation in end results. This also raises the philosophical question of whether it is fair to consider such design bugs as actual bugs. And whether we need to revisit what entails "correct" or "bug-free" in this context.

This paper describes the challenges and discusses some solutions for testing approximate computations from three perspectives across the computing stack:

- We first highlight the role of accuracy metrics for checking the accuracy at the application-level (Section II). The end-to-end perspective is important for any software or hardware system that leverages approximation.
- We describe software-level specifications of approximate functions and describe some approaches for accuracy-aware profiling and testing (Section III).
- We present a comprehensive hardware-focused study on voltage overscaling in arithmetic units, which discusses challenges for profiling and error recovery in approximate hardware (Section IV)

Finally, in Section V we discuss the set of challenges raised for the test community to consider as approximate computing becomes more prevalent.

## II. Catching Design Bugs under Approximation

**Quantifying Accuracy Loss:** Adapted from [2], Table I provides a first-order taxonomy for commonly used end-to-end accuracy metrics, according to the data type of end results, for a representative set of benchmark applications from Parsec [8],

| 2* **Class** | 2***Output Data Type** | 2***Accuracy Metric** | **Examples** | | |
|---|---|---|---|---|---|
| | | | **Application (Suite)** | **Domain** | **Metric** |
| 2*I | 2*Numeric: scalar | 2*Deviation [38] in output value | canneal (Parsec) | Optimization | Dev. in cost |
| | | | dedup (Parsec) | Compression | Dev. in file size |
| 4*II | 4*Numeric: multi- 4*dimensional | 4*Distortion [38] based | fluidanimate (Parsec) barnes, water (Splash2) | 2*n-body simulation | Dist. in "body" positions |
| | | | bodytrack (Parsec) particlefilter (Rodinia) | 2*Computer Vision | Dist. in coordinates |
| | | | cholesky, lu (Splash2) | Linear Algebra | Dist. in elements |
| | | | histo, tpacf (Parboil) | Histogram | |
| 4*III | 4*Compound | | streamcluster (Parsec) kmeans (Stamp) | 2*Clustering | Based on # mismatches |
| | | Based on # mismatches Positional error | UtilityMine (MineBench) | Data mining | |
| | | | ferret (Parsec) | Similarity search | |
| | | | radix (Splash2) | Sorting | Positional error |
| 3*IV | 3*Multi-media | Peak Signal to Noise Ratio (PSNR) | raytrace (Splash2) volrend (Splash2) | 2*Computer Vision | 3*PSNR, SSIM |
| | | Structural Similarity Index (SSIM) | x264 (Parsec) | Video Encoding | |

TABLE I
A FIRST-ORDER TAXONOMY FOR COMMONLY USED END-TO-END ACCURACY METRICS BY DATA TYPE OF APPLICATION OUTPUTS [2].

Splash2 [58], Rodinia [16], Parboil [53], MineBench [41], and Stamp [36] suites which feature noise tolerant algorithms. The taxonomy spans four classes.

For Classes I and II, the application output has a scalar or multi-dimensional numeric format. Common robust metrics represent variants of the *distortion* metric introduced by Misailovic et al. [38]. Distortion essentially is the average deviation (over all output elements) from the exact (non-approximate) value. Various methods exist to calculate the deviation per output value, and this is how variants of the distortion metric differ from each other. For example, if we use square of the difference between exact and approximate values to this end, the distortion metric becomes equivalent to mean square error.

Class III generates compound output data and spans clustering, mining, sorting and search applications. As a representative example, *ferret* from Parsec performs similarity search in an image database. For each of its input query images, the application finds a list of similar images (from the database) and ranks them by similarity. A robust metric needs to quantify the number of mismatches between the exact and approximate output image list in this case.

Class IV covers multi-media (i.e., image or video) output. Common robust metrics include the classic PSNR (Peak Signal to Noise Ratio) and the more recent SSIM (Structural Similarity Index) and its variants, which capture the human perception of accuracy loss better than PSNR [56].

Such end-to-end accuracy metrics, however, can only help quantify accuracy loss if the application outputs under approximation are *valid*. An invalid output does not necessarily imply an excessively corrupt output. As an example, let us consider *dedup* from Parsec, which performs file compression. Consulting Table I, we can use the deviation in the output file size as an accuracy metric. An invalid output under approximation in this case would be a corrupt file. The file size of such an invalid output can very well be equal to the file size of the exact (non-approximate) output. We therefore need an application-specific validity check. Simply trying to decompress the output file may suffice, as excessive corruption due to approximation may prevent successful decompression. Only if successful decompression is possible would calculation of the accuracy metric make sense.

Non-determinism represents yet another challenge, both due to the already probabilistic nature of the target algorithms, but also due to the wide spectrum of possible execution outcomes under approximation, depending on the specific approximation technique: no/catastrophic program termination, termination with invalid outputs, valid outputs spanning a wide range of accuracy loss, or some combination thereof. This brings a statistical aspect to testing and verification under approximation.

At the same time, both the size and the spread of input data values may have a huge impact on execution outcome and accuracy loss under approximation, which may further challenge the input vector generation and test coverage under approximation.

Last but not least, accuracy metrics cannot quantify acceptability of an execution outcome. The context, environment and conditions in which the corresponding application is deployed sets a threshold for acceptability. This may necessitate testing and verification by comparison to trade-off spaces or ranges, which span various acceptability levels, rather than to a specific point corresponding to a specific acceptability level.

We come back to the remaining critical issue of distribution/translation of end-to-end metrics to the interfaces of individual system components in a recursive or fractal fashion in Section III.

**Correctness Under Approximation:** End-to-end metrics cannot distinguish accuracy loss due to actual design bugs from accuracy loss due to approximation. When it comes to testing and verification of approximate software and/or hardware, design bugs can go undetected if their impact on the execution outcome is indistinguishable from what would correspond to the accuracy loss under approximation. If there was a way to prove that a design bug would behave always like this, for any input, it would arguably be safe to ignore such bugs – simply because they wouldn't have an impact on *correct* execution under approximation. Unfortunately, considering the challenges we discussed so far, this is a daunting task, particularly for test coverage. We need testing and verification methods that can distinguish the signature of a design bug in the application outputs from what would be *correct* per the stretched definition of *correctness* under approximation.

This brings us back to the question of what *correctness* entails under approximation. As a first step in this direction, the authors in [4] introduce the concept of *accuracy bugs*. The idea is stretching the definition of *correct* to include buggy but *approximately correct* parallel execution in the context of concurrency bugs, a rich set of severe bugs which encompasses data races, deadlocks, or atomicity/ordering/consistency violations. Accuracy bugs simply are concurrency bugs that do not cause program failures but manifest themselves as inaccuracy in outputs. Therefore, embracing accuracy bugs only comprises accuracy, but not *correctness*, as long as the corresponding loss in accuracy remains *acceptable*.

## III. Case Study I: Testing Approximate Software

*a) Running Example::* Let us consider a (simplifed) implementation of an algorithm that scales an image to a larger size. It consists of the function `scale` and the function `scale_kernel`. The function `scale` takes as input the scaling factor `f` (which increases the image size in both dimensions), along with integer arrays `src`, which contains the pixels of the image to be scaled, and `dest`, which contains the pixels of the resulting scaled image. The computation iterates over the image and calls the kernel, which interpolates the value of each pixel by a weighted sum of the neighboring pixels in the original image. The kernel computation can run on approximate hardware and produce approximate results:

```
int scale_kernel(float i, float j,
    int[] src, int s_height, int s_width);

void scale(float f, int[] src, int s_width, int s_height,
                    int[] dest, int d_height, int d_width)
{
  float si = 0, delta = 1 / f;

  for (int i = 0; i < d_height; ++i) {
    float sj = 0;
    for (int j = 0; j < d_width; ++j) {
      dest[IDX(i, j, d_width)] =
         scale_kernel(si, sj, src, s_height, s_width);
      sj += delta;
    }
    si += delta;
  }
}
```

The function `scale` takes as input the scaling factor `f` (which increases the image size in both dimensions), along with integer arrays `src`, which contains the pixels of the image to be scaled, and `dest`, which contains the pixels of the resulting scaled image. The algorithm calculates the value of each pixel in the final result by mapping the pixel's location back to the original source image and then taking a weighted average of the neighboring pixels, computed in `scale_kernel`.

*b)* **Kernel Reliability::** We can express the reliability of the computation using the specifications from [37]:

```
int<0.995 * R(i, j, src, s_height, s_width)>
scale_kernel (float i, float j, int[] src, int
              s_height, int s_width);
```

The specification of `scale_kernel` is a part of the type signature of the function. It denotes: (1) the *reliability degradation* of the function (0.995), representing the probability that the return value is correct if inputs were correct at the start, and (2) the joint probability, `R(i, j, src, s_height, s_width)`, that the inputs were computed correctly. One can similarly derive the specification for the `scale` function. Moreover, this specification can lead to the end-to-end accuracy specification, that the expected value of PSNR (after multiple executions of the program) is greater than $-10 \cdot \log_{10}(1 - r)$.

Next, we will describe some of the existing approaches for (1) inferring the specifications of the kernels and (2) if such specification exists, checking that the implementation likely satisfies the specification. The presentation in this section is derived from our previous publications [28], [37].

### A. Inferring Reliability Specifications

To specify the reliability of the kernel, the developer typically needs to relate the kernel's reliability with the program's end-to-end sensitivity metric. We show how to do it with *sensitivity profiling*. A sensitivity profiler takes (1) *an end-to-end sensitivity metric,* a function that compares the outputs of the original and approximate executions (e.g., PSNR); (2) the value of the sensitivity metric that characterizes the acceptable output quality; and (3) *sensitivity testing procedure,* which simulates errors in the outputs of the application.

A developer can write coarse-grained fault injection wrappers that inject noise into the computation. In general, the developer may use these wrappers to explore the sensitivity of the program's results to various coarse-grained error models. For `scale_kernel`, the developer can implement the following simple sensitivity testing procedure, which returns a random value for each color component.

Chisel's sensitivity profiler automatically explores the relation between the probability of approximate execution and the quality of the resulting image for the set of representative images. Conceptually, the profiler transforms the program to execute the correct implementation of `scale_kernel` with probability `r`, which represents the target reliability. The framework executes the faulty implementation `scale_kernel_with_errors` with probability `1-r`. The framework uses binary search to find the probability `r` for which the noisy program runs produce results with acceptable PSNR.

*a)* **Results::** Figure 1 presents a visual depiction of the results of scaling for different values of `r`. Note that implementations with low reliability (0.20-0.80) do not produce acceptable results. However, as `r` reaches values in the range of 0.99 and above, the results become an acceptable approximation of the result of the original (exact) implementation. The

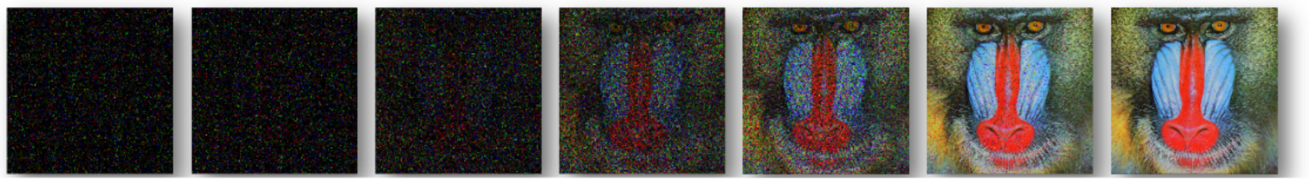

Fig. 1. Sensitivity Profiling for Image Scaling for Different Values of `r`. Illustration from [37]

target kernel reliability of `scale_kernel` of 0.995 tested on mulitple input images, yields an average PSNR of 30.9 dB.

*b) Outlook::* This general approach extends to other coarse-grained error scenarios, such as always returning a black or white pixel, or inverting the pixel computed by the original computation. Importantly, these models are *hardware-agnostic*, in that they do not require a detailed specification of the approximate hardware. They provide a lightweight method for estimating the sensitivity of the output of the program to the error in individual computations. Further refinements of this approach may include call-sensitive reliability specifications which would assign different accuracy specifications for each call site of the approximate function, or the or phase-sensitive specifications, which would assign different reliability to the kernel calls depending on whether it executes near beginning, in the middle, or near the end of the computation [39].

## B. Checking Reliability Specifications

Once the reliability specification for computational kernels (or more generally, any software component) exists, we can ask how to check whether the computation actually satisfies this specification. Interestingly, despite many rigorous specifications, a software developer who needs to implement, test, and tune these randomized algorithms and systems has virtually no tool support for this effort. Existing techniques only have support for binary notion of program correctness.

We recently developed AxProf, an algorithmic profiling framework for analyzing accuracy of approximate programs [28]. Given a high level accuracy specification (like the one we described in this section), AxProf constructs statistical models of accuracy, time, and memory, checks if any of them deviate from the algorithm specification, and if so, warns the developer.

AxProf supports probability query (which resembles reliability specification above). AxProf's specification language allows the specifications to be written in a form close to their mathematical counterparts, but enforces that they are specified in an unambiguous manner. For instance, a developer should explicitly write if a probabilistic specification is over inputs, items within an input, or runs. Such different cases may be handled more precisely with different statistical tests.

The key novelty of AxProf is the automatic generation of the accuracy checking code (which generates input data and invokes appropriate statistical tests) from a high-level probabilistic specification. For each specification, AxProf automatically (1) selects a proper statistical test, (2) generates checking code that extracts the output, aggregates the values, and applies the selected test, and (3) determines the number of runs to achieve a desired level of statistical confidence. In particular, the effectiveness of AxProf's analysis can be controlled by two knobs – statistical confidence or execution time of profiling.

Testing these programs often requires specific inputs, even though the specifications apply for all inputs. To automatically produce informative inputs, we provide several input generators for scalars, vectors, and matrices, that allow for various input properties to be modified: difference in the frequency of values (e.g., uniform vs skewed), order of data (different permutations), or various forms of correlations (e.g., the next value is a linear function of the previous). We present a new dynamic analysis that infers which of these properties affect the algorithm's accuracy the most.

**Results:** We used AxProf to check both kernel-level and application-level specifications. For the application-level specification, we check the decrease of PSNR as the subject of the reliability $r$ (the connection is derived in [37]). We use a set of representative inputs, requiring that the specification holds for all. For the kernel-level, we vary the set of pixels according to the pre-specified distribution and change its skew. In both cases, the properties should hold for all inputs, with high-probability over multiple runs. AxProf generates the code that calls the appropriate statistical tests and ensures that the specifications hold.

**Outlook:** The presence of approximations and random errors make testing programs that run on approximate hardware challenging. Tools that apply statistical testing provide confidence to the developers, who may have only basic statistical knowledge. The power of tools like AxProf is the ability to analyze systems of arbitrary complexity (in a way similar to statistical model checking [50]) and is agnostic of the method (hardware or software) that introduced noise in the execution. Complementary to testing reliability properties, researchers proposed various program verification techniques for program safety, e.g., ensuring that approximations do not cause fatal errors [13], [14].

## IV. Case Study II: HTM

For this case study, we describe *IgnoreTM*, a new framework for approximate computation that utilizes aggressive voltage scaling along with a novel error management scheme to improve energy efficiency. The key insight is that recovery from unacceptable errors may be facilitated by lightweight mechanisms adapted from hardware transactional memory (HTM) [24]. With the assistance of the HTM recovery mechanism IgnoreTM has been shown to improve energy efficiency significantly at a minimal run-time performance cost, while keeping errors with acceptable bounds.

Aggressively lowering supply voltage values can have the advantage of saving significant amounts of computing energy. However, lowering the voltage without scaling down operating frequencies may lead to intermittent timing errors (i.e., incorrect values at signal outputs due to signals not meeting their timing constraints). Several approaches have been proposed to manage these errors. For instance, Krause and Polian [31] allow errors due to voltage scaling to go uncorrected, given an application's inherent tolerance to errors. Varatkar and Shanbhag [55] employ algorithmic noise-tolerant blocks along with voltage over-scaling to correct for errors. Finally, ISA extensions are used in [49] and [21] to support approximate instructions that are executed on approximate functional units and storage. All the hardware-based approaches mention above either do not allow for error correction, or require special approximate hardware versions for the actual computation. Instead, what our ignoreTM approach proposes is an opportunistic way of dealing with errors that arise at runtime such that some errors may pass through without the need to waste runtime and energy to correct them. The hardware implementation itself remains exact; however, error recovery, when needed, is handled through our HTM-based scheme.

### A. The HTM Infrastructure

Hardware transactional memory was originally proposed as a means of recovering from data synchronization. Traditionally, the typical means of managing data consistency in shared memory systems is to use a *locking* mechanism, in order to guarantee that only a single transaction may modify a shared memory structure at any one time. That is, if a transaction needs to read/write to memory in a data structure, it must first obtain the lock and when it is done, it frees the locks so other transactions may access the shared structure. While this locking protocol enforces synchronization among transactions, it is inherently inefficient since it conservatively forces serial execution of the transactions even if there may be no read/write conflict for specific data within the shared memory structure. As an alternative, HTM optimistically allows transactions to proceeds with reads and writes to the data structure, and only if a true data conflict occurred, will the execution be aborted and retried. The HTM design is implemented using three key components:

1) a *bookkeeping* mechanism to keep track of read/write data accesses and detect conflicts,
2) a *data versioning* technique to keep track of original and speculative data versions for recovery in case of conflicts, and
3) a *check-pointing and rollback* mechanism to recover from data conflicts and retry failed transactions.

In the case of IgnoreTM, the idea is to use the HTM infrastructure not to manage data synchronization, but rather to manage timing errors induced from voltage overscaling. In this way, if a timing error is detected (through built-in error detection hardware) while executing a transaction, the system has a means of easily aborting the transactions (without involving the operating system), and re-executing the transaction to correct the error. Note that since we are not using HTM for memory synchronization, we may bypass the bookkeeping mechanism mentioned above, in order to obtain a more lightweight design.

We have implemented a version of this scheme in our prior work, called Edge-TM [43]. Adding safety margins (guard-bands) on the supply voltage prevents timing errors, but has a negative impact on performance and energy consumption. Edge-TM optimistically scales the voltage beyond the edge of safe operation for better energy savings and facilitates error recovery using HTM.

An underlying runtime system transparently manages the transactions with a core-level policy that optimistically lowers the voltage in small steps. A snapshot of the system state is taken before each transaction is started so that if the allowed error threshold is exceeded during transaction execution, this safe state can be restored. For timing error detection, we assume that each core is equipped with runtime error-detection circuitry, such as error-detection sequential (EDS) [10].

The Edge-TM policy utilizes a combination of static and dynamic monitors to determine appropriate conditions for voltage adjustment such that timing errors do not become so frequent such that all the energy gains made from operating at a reduced voltage are undone by the need to repeatedly execute transactions that incurred timing errors.

To allow for improved runtime and error efficiency, our new IgnoreTM approach builds upon the EdgeTM infrastructure. Now, instead of always aborting when an error is detected, the hardware can opportunistically *ignore the timing error*, thereby removing the need to abort and re-execute a transaction after *every* error. The added challenge for IgnoreTM is to determine when a timing error can be ignored, thereby avoiding re-execution. This determination is made with the help of profiling an application at runtime, using appropriate error models.

As a first step, we focus on floating-point applications, many of which are naturally tolerant to certain types of inaccuracies and errors. We estimate likely erroneous values using a Critical Bit model [57]. The Critical Bit model integrates computation history and value correlation with bit-wise dependency by identifying the critical bit of the functional unit architecture. The critical bit corresponds to a signal that is along the critical path of a particular arithmetic operation.

By profiling various floating point applications using the critical bit model we are able to accurately gauge how timing errors manifest themselves on the result of floating point operations (i.e., addition or multiplication). After profiling, instructions that are determined to be amenable to approximation are marked "approximatable" and an overall error rate threshold is determined according to a user specified accuracy loss tolerance. Note that a tool such as AxProf (as described in Section III) may be used to automatically generate the accuracy checking code. However, by using our Critical bit model, we incorporate some knowledge of the hardware implementation into the simulation of the checking code, which allows for a more accurate accounting for timing errors.

We have shown that *IgnoreTM* achieves up to 47% total energy savings without impacting runtime [42]. Moreover, our approach allows for an additional 13-18% energy savings compared to existing voltage scaling techniques such as Edge-TM [43] that do not apply approximations but always correct timing errors, while either improving or having a negligible impact on runtime.

## B. Implementation Details

We handle timing errors at the granularity of a transaction. That is, we enclose within a transaction each block of the program that has instructions that we want monitored for timing errors. This means that we protect with transactions every part of the program where voltage scaling will be applied, regardless of whether it will be approximated or not. We then monitor the actual error rate in real time, as the program is being executed. If the error threshold is not exceeded (i.e., timing errors are occurring at a rate that by profile was determined to be too high to retain acceptable accuracy loss), the transaction *commits*, the checkpoint is discarded, and speculative changes to the data become permanent. If the error rate threshold is exceeded, the transaction *aborts*, and a *rollback* mechanism restores the internal core state. In addition, data are restored to their original values and speculative copies are discarded.

We use a distributed logging scheme to enable data versioning. Logs are distributed among the cache memory banks and each bank keeps a fixed-size log space for each core in the system. Note that only the first time an address is written does its original value need to be saved in the log, so the log size is quite modest. The log saving and restoration process is done independently at each memory bank and does not require interaction with other banks, which makes it very fast and efficient. Details of our data versioning implementation may be found in [43].

Voltage adjustments are determined based on the number of completed transactions between aborts (i.e., the number of *consecutive commits*). In our *IgnoreTM* policy, we take into account the frequency of a timing violation during a transaction that has to be corrected, and its effect on the total system energy consumption. When such frequency of a timing violation is too high (thus increasing the number of aborts and total energy consumption), we increase our operating voltage in order to save energy by lowering the abort rate. The *IgnoreTM* policy balances ignoring timing violations, correcting timing violations, and changing voltage levels to optimize energy savings while maintaining acceptable program output accuracy.

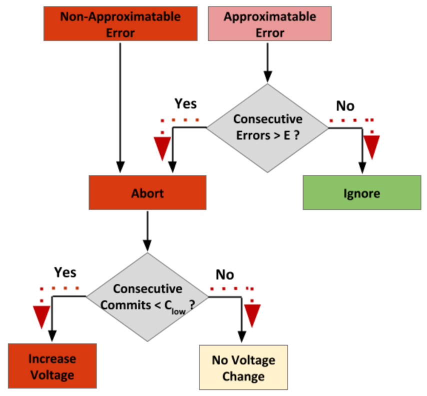

Fig. 2. Flow diagram of *IgnoreTM* DVS Policy showing voltage adjustment policy for timing violations [42].

Figure 2 shows the flow diagram for our voltage adjustment policy due to timing errors. The policy makes decisions on voltage adjustment on two occasions, (1) when a transaction successfully commits, and (2) when a transaction aborts, before it is re-executed. A voltage adjustment decision is based on the number of consecutive commits, $C$, that are experienced and the error threshold, $E$, (i.e., the maximum number of ignored timing errors in a single run of a transaction), which determines the level of approximation the transaction will tolerate. Since every benchmark and every use case of each benchmark can have different error thresholds, the user can set a threshold for the number of ignored timing errors *per each transaction*.

## V. Putting It All Together: Challenges Across the System Stack

We next describe several challenges for testing approximate applications at different levels of the system stack.

### A. Distinguishing Actual Bugs from Approximation-Induced Errors

The key challenge in testing approximate programs is providing statistical guarantees. Actual design bugs, unfortunately, are at least as non-deterministic in nature as execution outcome under approximation. Defining equivalence classes (of actual bugs and approximation-induced errors) based on system-level manifestation hence becomes very difficult. Intuitively we would expect testing to become easier if we could identify such equivalence classes. This is simply because we could exclude equivalent design bugs from testing and verification. No such simplification would be valid though, unless we prove equivalence over all possible execution scenarios. The search space for pair-wise comparison (to prove equivalence) is huge and complex due to non-determinism on both ends.

## B. Classic Hardware Testing as an Enabler for Disciplined Approximation

Our discussion so far has focused on opportunities and challenges when it comes to the testing and verification of approximate computing systems. The rich knowledge base from the testing community can also serve as an enabler for disciplined approximate computing. In the end, we can regard each point of approximation – be it in hardware or software in space and time – essentially as a point of fault injection. Hence, to quantify system level implications of approximation, including but not limited to the impact on computational accuracy, we can practically rely on classic fault detection and propagation analysis. Input test vector generation and coverage analysis can further help in bounding accuracy loss by providing statistical guarantees. Moreover, we can deploy classic online testing techniques to support runtime approximation, in order to keep approximation induced errors at bay. There are numerous opportunities to be explored.

## C. Error Detection and Program Adaptation

The testing approaches discussed so far are performed *offline*, during the design and development phases of approximate hardware and software. While these techniques do not incur execution overhead during the program execution, they may be insufficient if e.g., input profile of the application or the assumptions about system change during execution. We may ask how can we apply the techniques for error detection and recover *online*, during the execution of an approximate program. Runtime error identification and recovery has a long history in reliability engineering, where the source of errors are soft faults or component failures (e.g., see [23], [44], [45] for detection and [19], [20], [22], [34] for recovery), but the topic has been less studied for general approximations.

In approximate computing, errors happen with much larger frequency and have a different error profile than soft faults. Therefore, the systems should be able to monitor errors, and if necessary constantly adapt to the changes in performance and/or accuracy. The existing research presents several techniques for runtime adaptation of software, e.g., [6], [25]–[27] and error estimation for individual inputs [32], [60]. However, the questions about generality and seamless integration of existing approaches with compilers and hardware interfaces (especially in the emerging heterogeneous systems) remain open, and a topic for future research.

## D. The Hardware-Software Boundary

End-to-end testing of approximate system is challenging today in part because it is difficult to integrate all aspects of approximation at different levels of the computing stack. To systematically analyze approximate systems, one promising direction is to find abstract interfaces between the layers of the system stack that will allow decoupling the reasoning between these layers. The reliability specification from Section III is one example of decoupling. The hardware architects who design accelerators can check that their designs satisfy such specifications and use some means of error recovery if they are not, perhaps by using error mitigation techniques such as *IgnoreTM* (Section IV). The software developer can separately ensure that the program satisfies the end-to-end accuracy metric from Section II.

The models of approximate hardware are abstractions that describe the accuracy and performance implications of approximate operations. They extend the instruction set architecture (ISA) with approximation-related annotations and guarantees. Note that the operations may be fine-grained (e.g., approximate addition) or coarse grained (e.g., approximate FFT). The models can thus represent the error frequency, magnitude, or another instruction-level accuracy metric. Due to the nature of the approximations, the models may be both conservative (e.g., the error will always be below the threshold) or empirical (e.g., the average error over the input space is below the error). The existing models provide several simple abstractions [12], [37], [49], [54], [57], but they can capture only a limited set of error mechanisms. Despite some recent work on flexible abstractions to support application verification [9], a key question is how to design expressive error and performance models that can benefit both hardware and software communities.

## E. Monitoring and Profiling

As we have discussed in previous sections, profiling is a critical component of determining accuracy loss and intrinsic noise tolerance. Tools such as AxProf [28] can go a long way in streamlining the process of generating meaningful accuracy checking code. Indeed such code could also be used for more general design bug testing and verification aids. While AxProf is hardware agnostic, it will be important to consider how faults or other types of noise are manifested in the output. This will ultimately require some underlying understanding of the hardware implementation and more precisely how the approximation will be manifested in the hardware. Accurate approximation models will have to be developed, but how accurate do they need to be and how generalizable can they be from one hardware implementation to another? Finally, as much as profiling will be indispensable in helping to evaluation approximation loss, how much do designers want to rely on profiling alone? What are the possible alternatives in the future?

## Acknowledgments

The authors would like to thank Ismail Akturk, Karen S. Khatamifard, Dimitra Papagiannopoulou and Tali Moreshet for their help in developing some of the ideas presented in this paper.

## 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*. ACM, 2015, pp. 711–730.

[2] I. Akturk, K. Khatamifard, and U. R. Karpuzcu, "On Quantification of Accuracy Loss in Approximate Computing," in *12th Annual Workshop on Duplicating, Deconstructing and Debunking (WDDD)*, June 2015.

[3] I. Akturk, N. S. Kim, and U. R. Karpuzcu, "Decoupling Control and Data Processing for Approximate Near-threshold Voltage Computing," *IEEE Micro Special Issue on Heterogeneous Computing*, 2015.

[4] I. Akturk, R. Akram, M. M. Islam, A. Muzahid, and U. R. Karpuzcu, "Accuracy Bugs: A New Class of Concurrency Bugs to Exploit Algorithmic Noise Tolerance," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016.

[5] D. S.-G. U. R. K. Alexandra Ferrerón, Jesús Alastruey-Benedé, "AISC: Approximate Instruction Set Computer," *Workshop on Approximate Computing in conjunction with ASPLOS*, March 2018.

[6] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 198–209.

[7] F. Betzel, S. K. Khatamifard, H. Suresh, D. J. Lilja, J. Sartori, and U. R. Karpuzcu, "Approximate Communication: Approximation Techniques for Communication Reduction in Parallel Systems," *ACM Computing Surveys*, vol. 51, no. 1, January 2018.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," Princeton University, Tech. Rep. TR-811-08, January 2008.

[9] B. Boston, Z. Gong, and M. Carbin, "Leto: verifying application-specific hardware fault tolerance with programmable execution models," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 163, 2018.

[10] K. Bowman, J. Tschanz, S. Lu, P. Aseron, M. Khellah, A. Raychowdhury, B. Geuskens, C. Tokunaga, C. Wilkerson, T. Karnik, and V. De, "A 45nm resilient microprocessor core for dynamic variation tolerance," *JSSC*, vol. 46, no. 1, pp. 194–208, Jan 2011.

[11] M. Breuer, "Hardware That Produces Bounded Rather Than Exact Results," in *Design Automation Conference (DAC)*, June 2010.

[12] M. Carbin, S. Misailovic, and M. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware, 2013," in *28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA/SPLASH 2013), Indianapolis, IN, USA*, 2013.

[13] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *PLDI*, 2012.

[14] ——, "Verified integrity properties for safe approximate program transformations," in *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*, 2013.

[15] S. T. Chakradhar and A. Raghunathan, "Best-effort Computing: Rethinking Parallel Software and Hardware," in *Design Automation Conference (DAC)*, 2010.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[17] V. Chippa, D. Mohapatra, and A. Raghunathan, "Scalable Effort Hardware Design: Exploiting Algorithmic Resilience for Energy Efficiency," *Design Automation Conference (DAC)*, June 2010.

[18] H. Cho, L. Leem, and S. Mitra, "ERSA: Error Resilient System Architecture for Probabilistic Applications," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 4, 2012.

[19] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An Architectural Framework for Software Recovery of Hardware Faults," in *International Symposium on Computer Architecture (ISCA)*, 2010.

[20] M. A. de Kruijf, K. Sankaralingam, and S. Jha, "Static analysis and compiler design for idempotent processing," in *ACM SIGPLAN Notices*, vol. 47, no. 6. ACM, 2012, pp. 475–486.

[21] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture Support for Disciplined Approximate Programming," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[22] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," 2010.

[23] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost Program-level Detectors for Reducing Silent Data Corruptions," 2012.

[24] M. Herlihy and E. Moss, "Transactional memory: architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 289–300, May 1993.

[25] H. Hoffmann, "Jouleguard: energy guarantees for approximate applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 198–214.

[26] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats for software performance and health," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 347–348, 2010.

[27] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *ACM SIGPLAN Notices*, vol. 46, no. 3. ACM, 2011, pp. 199–212.

[28] K. Joshi, V. Fernando, and S. Misailovic, "Statistical algorithmic profiling for randomized approximate programs," in *ICSE*, 2019.

[29] U. Karpuzcu, I. Akturk, and N. S. Kim, "Accordion: Toward Soft Near-Threshold Voltage Computing," in *International Symposium on High Performance Computer Architecture (HPCA)*, February 2014.

[30] S. K. Khatamifard, I. Akturk, and U. R. Karpuzcu, "On Approximate Speculative Lock Elision," *IEEE Transactions on Multiscale Computing Systems, Special Issue on Emerging Technologies and Architectures for Manycore Computing*, November 2017.

[31] P. Krause and I. Polian, "Adaptive voltage over-scaling for resilient applications," in *DATE*, March 2011.

[32] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang, "Input responsiveness: using canary inputs to dynamically steer approximation," *PLDI*, 2016.

[33] X. Li and D. Yeung, "Application-Level Correctness and its Impact on Fault Tolerance," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[34] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler directed lightweight soft error resilience," in *ACM Sigplan Notices*, vol. 50, no. 5, 2015, p. 2.

[35] J. S. Miguel, M. Badr, and N. E. Jerger, "Load Value Approximation," in *International Symposium on Microarchitecture (MICRO)*, December 2014.

[36] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[37] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels," in *OOPSLA*, 2014.

[38] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of Service Profiling," in *International Conference on Software Engineering (ICSE)*, 2010.

[39] S. Mitra, M. K. Gupta, S. Misailovic, and S. Bagchi, "Phase-aware optimization in approximate computing," in *CGO*, 2017.

[40] T. Moreau, J. San Miguel, M. Wyse, J. Bornholt, A. Alaghi, L. Ceze, N. Enright Jerger, and A. Sampson, "A taxonomy of general purpose approximate computing techniques," *IEEE Embedded Systems Letters*, vol. 10, no. 1, pp. 2–5, March 2018.

[41] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "MineBench: A Benchmark Suite for Data Mining Workloads," in *International Symposium on Workload Characterization*, October 2006.

[42] D. Papagiannopoulou, S. Whang, T. Moreshet, and R. I. Bahar, "Ignoretm: Opportunistically ignoring timing violations for energy savings using htm," in *Proceedings of the IEEE/ACM Design Automation and Test in Europe Conference (DATE)*, 2019.

[43] D. Papagiannopoulou, A. Marongiu, T. Moreshet, M. Herlihy, and R. I. Bahar, "Edge-TM: Exploiting transactional memory for error tolerance and energy efficiency," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, pp. 153:1–153:18, Sep. 2017.

[44] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software Implemented Fault Tolerance," in *Proc. of Intl. Symp. on Code generation and optimization*. Washington, DC, USA: IEEE Comp. Society, 2005.

[45] G. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Design and evaluation of hybrid fault-detection systems," 2005.

[46] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener, "Programming with Relaxed Synchronization," in *ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012.

[47] M. Rinard, "Parallel Synchronization-Free Approximate Data Structure Construction," *5th USENIX Workshop on Hot Topics in Parallelism*, 2013.

[48] M. C. Rinard, "Using Early Phase Termination to Eliminate Load Imbalances at Barrier Synchronization Points," in *Conference on Object-oriented Programming Systems and Applications (OOPSLA)*, 2007.

[49] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-power Computation," in *Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[50] K. Sen, M. Viswanathan, and G. Agha, "Statistical model checking of black-box probabilistic systems," in *International Conference on Computer Aided Verification*, 2004.

[51] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing Performance vs. Accuracy Trade-offs with Loop Perforation," in *European Software Engineering Conference and European Conference on Foundations of Software Engineering (ESEC/FSE)*, 2011.

[52] P. Stanley-Marbell, A. Alaghi, M. Carbin, E. Darulova, L. Dolecek, A. Gerstlauer, G. Gillani, D. Jevdjic, T. Moreau, M. Cacciotti, A. Daglis, N. D. E. Jerger, B. Falsafi, S. Misailovic, A. Sampson, and D. Zufferey, "Exploiting errors for efficiency: A survey from circuits to algorithms," *CoRR*, vol. abs/1809.05859, 2018.

[53] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," University of Illinois at Urbana-Champaign, Urbana, Tech. Rep. IMPACT-12-01, Mar. 2012.

[54] G. Tziantzioulis, A. M. Gok, S. M. Faisal, N. Hardavellas, S. Ogrenci-Memik, and S. Parthasarathy, "b-HiVE: A bit-level history-based error model with value correlation for voltage-scaled integer and floating point units," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[55] G. Varatkar and N. Shanbhag, "Error-resilient motion estimation architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (TVLSI)*, vol. 16, no. 10, pp. 1399–1412, Oct 2008.

[56] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, April 2004.

[57] S. Whang, T. Rachford, D. Papgiannopoulou, T. Moreshet, and R. I. Bahar, "Evaluating critical bits in arithmetic operations due to timing violations," in *IEEE High Performance Extreme Computing Conference*, Sept. 2017.

[58] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *International Symposium on Computer Architecture (ISCA)*, 1995.

[59] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design Test*, vol. 33, no. 1, pp. 8–22, Feb 2016.

[60] R. Xu, J. Koo, R. Kumar, P. Bai, S. Mitra, S. Misailovic, and S. Bagchi, "Videochef: efficient approximation for streaming video processing pipelines," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 43–56.

[61] T. Y. Yeh, G. Reinman, S. J. Patel, and P. Faloutsos, "Fool Me Twice: Exploring and Exploiting Error Tolerance in Physics-based Animation," *ACM Transactions on Graphics*, vol. 29, December 2009.

[62] S. Yesil, I. Akturk, and U. R. Karpuzcu, "Toward Dynamic Precision Scaling," in *IEEE Micro Special Issue on Approximate Computing*, July/August 2018, in press.