Input Responsiveness: Using Canary Inputs to Dynamically Steer Approximation

Michael A. Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars and Lingjia Tang
Department of Electrical Engineering and Computer Science, University of Michigan, USA
{mlaurenz, parkerhh, mehrzads, mahlke, profmars, lingjia}@umich.edu

Abstract

This paper introduces Input Responsive Approximation (IRA), an approach that uses a *canary input* — a small program input carefully constructed to capture the intrinsic properties of the original input — to automatically control how program approximation is applied on an input-by-input basis. Motivating this approach is the observation that many of the prior techniques focusing on choosing *how to approximate* arrive at conservative decisions by discounting substantial differences between inputs when applying approximation. The main challenges in overcoming this limitation lie in making the choice of how to approximate both effectively (e.g., the fastest approximation that meets a particular accuracy target) and rapidly for every input. With IRA, each time the approximate program is run, a canary input is constructed and used dynamically to quickly test a spectrum of approximation alternatives. Based on these runtime tests, the approximation that best fits the desired accuracy constraints is selected and applied to the full input to produce an approximate result. We use IRA to select and parameterize mixes of four approximation techniques from the literature for a range of 13 image processing, machine learning, and data mining applications. Our results demonstrate that IRA significantly outperforms prior approaches, delivering an average of 10.2× speedup over exact execution while minimizing accuracy losses in program outputs.

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908087

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Runtime environments; D.3.3 [*Programming Languages*]: Language Constructs and Features – Input/output; C.0 [*General*]: Hardware/software interfaces

General Terms Performance, Experimentation, Design

Keywords Performance, Runtime Systems, Compilers, Approximate Computing

1. Introduction

Emerging applications in the domains of image and sound processing, computer vision, machine learning, and data mining are significantly increasing the processing demands on compute infrastructure as the adoption of smart technologies like intelligent virtual assistants [4, 21, 36] and wearable devices [20, 35] rises. These emerging applications rely heavily on *regularly-structured computations* on inputs that include images, video, and sound, and have loose constraints on the quality of output. The need for significant improvements in processing throughput for these classes of applications along with loose quality constraints make them ideal candidates for *approximate computing*, where small amounts of output accuracy can be traded for large improvements in performance or energy.

The general purpose approximate computing techniques typically applied to regularly-structured computations, such as loop perforation [27, 45], algorithm selection [3, 15], and numerical approximation [25], have been important and successful vehicles for realizing approximation in practice. These approaches can be realized on commodity hardware, apply to a variety of problem types, and are straightforward for programmers to implement. A central question in approximate computing is *how to approximate*, i.e., how to configure and parameterize an approximate program to yield the right tradeoff between performance and accuracy. Prior work focusing on the problem of choosing how to approximate has shown modest performance improvements of 1.1× to 4× [6, 27, 28, 47, 49, 50, 56], often relying on *calibration* or *profiling* to choose how to approximate.

Calibration computes both exact and approximate results, then compares them to measure the accuracy of the approximate approach on some (set of) program inputs. Calibration has been used to drive offline approaches [27], runtime systems [49], and within systems that use a combination of the two [6]. Because it is expensive to compute the exact solution and the accuracy of the approximate solution(s) on every input, calibration must be used sparingly, encumbering the flexibility of approximation and ultimately the performance gains that can be realized. Profile guided approaches make approximation decisions based on average- or worst-case input behavior [18, 28, 56]. These techniques rely on training with inputs that are representative of real-world inputs, which may be difficult to achieve in practice.

The motivation for this work is that, common to both classes of approaches, one approximation is used to cover multiple inputs. Focusing on worst case accuracy can result in overly conservative approximation for many inputs, while focusing on average case accuracy may be overly aggressive and fail to deliver sufficient accuracy in the worst case. As we show in this work, designing approximation systems that discount the differences between inputs hinders both the performance and accuracy of software-based approximation.

This paper presents an approach to addressing this limitation that is guided by two observations. Firstly, the accuracy of approximate programs can depend heavily on program input. Consider the example presented in Figure 1, which shows two images that have been processed by an identical approximate gamma correction with results that are of wildly different quality. Typical approaches to dealing with this difference in quality would dial down the aggressiveness of the approximation for both images, sacrificing performance for the first to produce satisfactory accuracy for the second. Secondly, regularly-structured computations have (1) *data-centric computation*, where the amount of computation depends on the size of the input data, such as is the case in many iterative and recursive algorithms in data mining, image processing, and machine learning and (2) *summarizable inputs*, where there may be redundancy or patterns in the input data such that the input's characteristics can be concisely represented. While these characteristics are not universal across all programs (for example, a compiler's input or a database schema have deep structure that can be easily broken), they appear often in the classes of programs that are candidates for approximation. On such programs, our insight is that a small input can be used to quickly learn the accuracy and performance characteristics of approximations without relying on calibration or profiling.

This paper introduces *Input Responsive Approximation* (IRA), a runtime approximation system designed to leverage this insight in order to dynamically and automatically configure the approximation options for each program input, including selecting which code regions to approximate and how to tune the approximations within those regions. IRA achieves this by creating a *canary input* — a much smaller representation of the full input — at the outset of the program. The canary input is used to dynamically predict the accuracy and speedup characteristics of the full input for a number of approximation options, then to choose the fastest option that achieves the desired level of accuracy. The specific contributions of this work are:

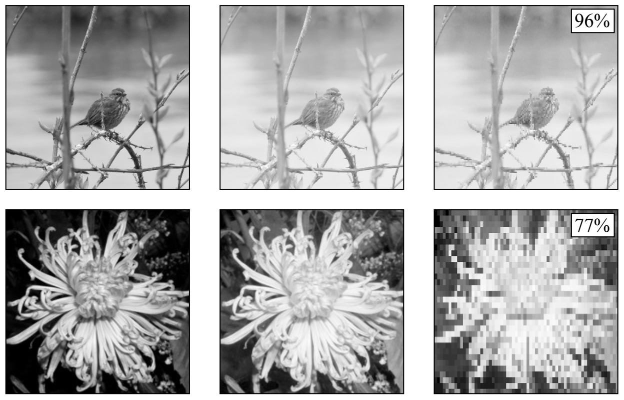
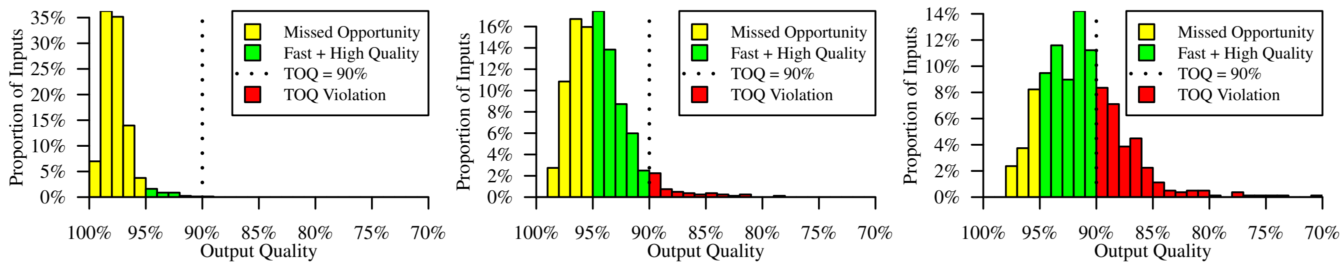


Figure 1. One approximation approach (16×8 tiling [50]) produces outputs of very different quality across inputs

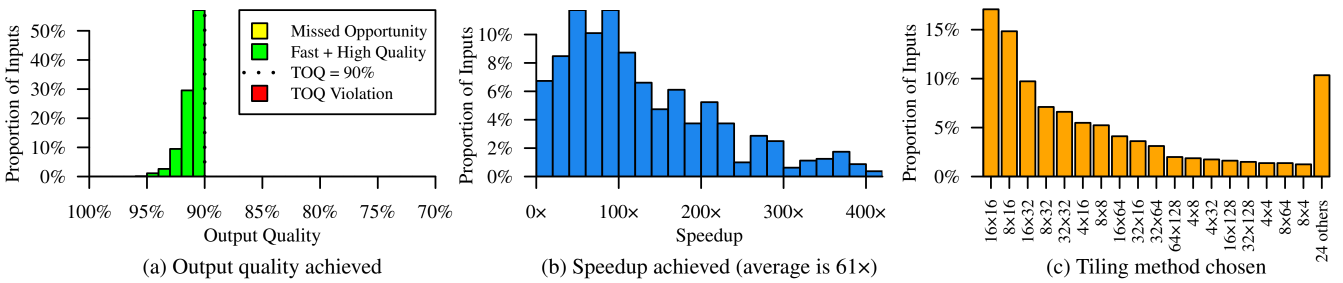
- **Canary Inputs –** we describe a methodology for reducing the inputs to many regularly-structured computations to *canaries* – smaller representations of the full inputs – that can used to predict program performance and accuracy when subjected to various approximations. Canary creation is guided by statistical hypothesis testing, yielding canaries that are statistically guaranteed to share key properties with the full input (§4.1).
- **Canary-driven Approximation Runtime System –** we describe the design and implementation of IRA, a runtime system that uses canary inputs to automatically configure approximation for every input supplied to a program (§4). IRA determines where to approximate, automatically selecting which code regions are most amenable to approximation for each input, as well as configuring the approximation within those regions to the fastest configuration that meets a specified accuracy bound (§4.2 and §4.3).
- **Effective, Automatic Approximation –** we evaluate IRA on real server hardware, demonstrating that it produces effective input responsive approximations. The evaluation incorporates four well-known classes of approximate techniques from the literature: loop perforation [27, 45], tiling [50], algorithm choice [3, 15] and numerical approximation [25] within 13 image processing, machine learning data mining and computer vision applications (§5).

The overhead of using IRA to create and use canaries to dynamically search for approximations averages 3.2% of exact execution time. Even with this overhead included, IRA



(a) 4×2 tiling approximation (5.9× speedup) (b) 8×8 tiling (22× speedup) (c) 16×16 tiling (83× speedup)

Figure 2. Histograms of the accuracy of three tiling approximations applied to the same 800 images; some mix of missed opportunities and unacceptably low accuracy are present in each approximation



(a) Output quality achieved (b) Speedup achieved (average is 61×) (c) Tiling method chosen

Figure 3. A dynamic oracle approximation system using the most effective tiling approximation method (fastest without violating TOQ) achieves an average speedup of 61× and uses 42 different approximation options

results in an average speedup of 10.2× across our 13 test applications with small accuracy losses in program outputs. IRA outperforms oracle versions of two classes of prior work while delivering the same level of accuracy, achieving more than 4× the speedup of oracle calibration techniques and 2× the speedup of oracle static profiling.

2. The Case for Input Driven Dynamism

The ability of approximate computation to produce high-quality results is one of the keys to making approximation broadly deployable in real systems. Many techniques for preserving result accuracy when choosing how to approximate focus on the worst case, resulting in overly conservative approximation for other cases. Here we discuss the opportunity available in the presence of a technique that dynamically controls approximation for individual inputs.

2.1 Input Matters for Output Quality

Input is an important part of the accuracy of an approximate computation. To illustrate this, we detail the output quality produced by three different tiling approximations [50] of an image processing application called *gamma correction* [42] applied to 800 input images. Tiling is based on the assumption that, in many application domains such as image and video processing, elements nearby one another (e.g., pixels in an image) are likely to have similar values. Instead of computing each element of the output, a tiling approximation computes a single output element and projects that output onto the surrounding elements to form a tile. Tiling can be tuned to trade off lower accuracy for better performance by increasing the size of the tile.

Figure 2 presents histograms of the output quality for 800 different images across three tile sizes. For the purposes of illustration, we assume that the *target output quality*[1] (TOQ) of the approximation is 90%. As shown in the figure, for all three tile sizes, different inputs can result in very different output qualities. For example, across these inputs 8×8 tiling (Figure 2(b)) results in output qualities ranging from 78%-99% because the assumption made by the approximation technique (that nearby pixels are similar to one another) holds true to a different extent depending on the composition of the input. Furthermore, we have observed that a wide range in output quality across inputs is not unique to tiling approximation and gamma correction, occurring across many applications and approximation techniques.

2.2 Limitations of Conventional Approaches

A common approach used to choose how to approximate is to select a single approximation option for some program and apply that approximation to multiple inputs. This approach to approximation suffers from a form of the *problem of aggregation*, in which aggregate behavior (average or

[1] Target output quality (TOQ) is the minimum acceptable result accuracy [49], supplied by the user of the application.

worst) is not necessarily representative of individual behavior. In the presence of multiple differing inputs, an approximation system that uses a single approximation across inputs either leaves performance opportunities on the table, violates output quality restrictions, or both.

To illustrate this, we refer again to Figure 2, where (a), (b) and (c), are histograms of result accuracy for three increasingly aggressive approximate gamma corrections applied to 800 input images. We assume a TOQ of 90%, and characterize the outputs as falling into 3 classes: TOQ violating approximations (< 90% output quality), fast + high quality approximations (90-95% output quality) and missed opportunities (95-100% output quality). The 4×2 tiling approximation, shown in 2(a), produces minimal TOQ violations, but the speedup is limited to 5.9×. The bulk of these output qualities can be classified as missed opportunities. A more moderate approach, 8×8 tiling, is shown in 2(b). In this case, 5% of the results violate the TOQ with a speedup of 22×. Finally, the results of an aggressive approximation are shown in 2(c). This approach uses 16×16 tiling and yields 83× speedup with 30% of the outputs violating the TOQ.

2.3 The Opportunity for Dynamism

Ideally, approximation would have the best of both worlds – no missed opportunities and no TOQ violations. This could be achieved by dynamically choosing the most *effective* approximation for each input – the fastest approximation that does not violate the TOQ. Figure 3(a) presents a histogram of output quality over the 800 inputs when using a dynamic oracle to choose the most effective option from a wide range of tiling approximations. Unlike the previous example, the most effective approximation is always fast and high quality, never leaving performance on the table and never violating the target output quality.

Moreover, Figure 3(b) shows a histogram of the speedups achieved on the set of 800 inputs. The speedups vary significantly, ranging from 3.5× to 410× (average 61×) due to the fact that a wide range of approximations are chosen. As shown in Figure 3(c), across 800 inputs, 42 unique approximation methods are chosen, with no single approximation being used on more than 17% of the inputs. That is, a wide range of approximation methods are used to obtain the maximally effective approximation across the set of inputs and no single approximation is dominant. *The key to taking advantage of this opportunity is to customize the approximation for each input on an individual basis, and to develop that customized approximation quickly.*

3. Overview of Approach

Given a program, a menu of possible approximation options, and an input to the program, the goal of *Input Responsive Approximation* (IRA) is to rapidly choose an effective approximation for that input. Our approach to achieving this goal is shown in Figure 4 and does the following:

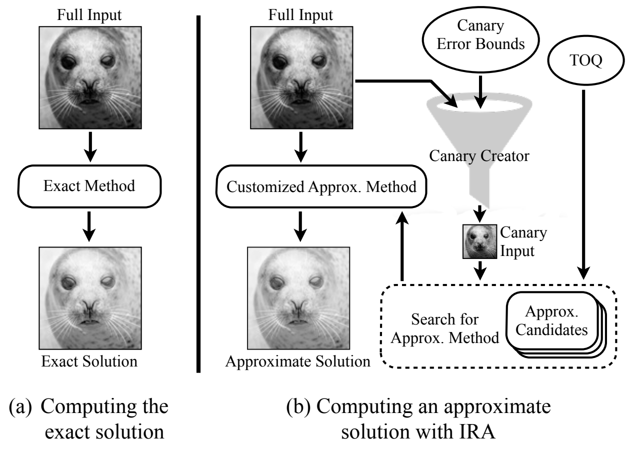


Figure 4. Exact computation and approximation with IRA

1. **Canary Input –** first, IRA dynamically produces a *canary input*, a smaller representation of the input (§4.1). The creation of the canary is guided by hypothesis testing, a statistical framework used to ensure that the resulting canary is large enough to be representative of the full input, sharing key properties with the full input, while being no larger than necessary to minimize the overhead of leveraging the canary.
2. **Customize the Approximation –** next, an exact solution and multiple approximate solutions are computed using the canary to select the most effective from among the available approximations, including selecting the code regions to approximate and how to approximate within those regions (§4.2). Because the canary input is much smaller than the full input, IRA is able to rapidly forecast how numerous approximations fare on a particular input by running the canary input with each of those approximations. Unlike prior work, IRA predicts the accuracy and performance of approximations on each input on demand and *ex ante*, allowing it to choose a customized, effective approximation for every input.
3. **Compute Approximate Solution –** finally, the customized approximation deemed effective for the canary is applied to the full input to produce an approximate solution that is of acceptable accuracy (§4.3). As we show later, this approach is extremely effective, leading to large performance improvements with small accuracy losses.

4. IRA Design and Implementation

This section provides a detailed description of how IRA chooses and leverages an approximation that is customized and effective for each program input.

4.1 Reasoning About Canary Inputs

A canary is a small program input, crafted out of the full program input to reflect the full input's properties. A plausible approach for creating canaries could be to simply sample down every full input at the same rate. Unfortunately, this approach produces canaries that are either (1) larger than necessary for well-behaved inputs, introducing extra overhead in the approximation search process or (2) too small to adequately represent the full input, resulting in a search that provides a misleading model of approximation accuracy characteristics. Instead, we have observed that a dynamic approach to producing canaries helps avoid these problems, where the canary can be chosen to be both as small as possible and large enough to maintain sufficient similarity to the full input. We experimentally explore this issue further in §5.2.

4.1.1 Challenges

Dynamically creating a canary input that exhibits the properties of the full input has three main challenges. First, in determining the similarity of the canary and the full input, we require a definition of similarity that reflects meaningful input properties. Second, we must be able to choose the canary in a way that is both computationally inexpensive and ensures that the definition of similarity is satisfied. Third, we want to choose a canary that is much smaller than the full input, as this will be a large determinant of the time spent employing the canary to test various approximations.

To address the first challenge, this work defines and explores four different metrics of similarity, designed to span a range of definitions of what it means for inputs to be similar to one another. These metrics range from a very simple metric designed to ensure that the values in the canary are close, on average, with the values in the full input to more sophisticated, complex metrics that ensure the similarity of local properties within small regions of the input.

To address the second challenge, we ensure low overhead in the canary creation process by leveraging statistical sampling in the analysis of each potential canary input, allowing similarity metrics to be computed on just a small subset of the canary input when analyzing its similarity to the full input. To ensure that the definition of similarity is satisfied in a chosen canary, we use a carefully designed algorithm based on robust, automated hypothesis tests that minimize the likelihood of making an incorrect decision about each canary. In particular, we take special care to design our approach to avoid both *false negatives* – incorrectly finding dissimilarity – and *false positives* – incorrectly finding similarity. These are also known as Type I and Type II errors, respectively. The avoidance of false positives ensures that the canary we select is highly likely to be similar to the full input.

On the other hand, avoiding false negatives so that the chosen canary is no larger than necessary is key to ensuring that the third challenge is addressed. If we mistakenly rejected a small canary that was actually similar to the full input in favor of a larger canary, the canary-driven search can have unnecessarily high overhead.

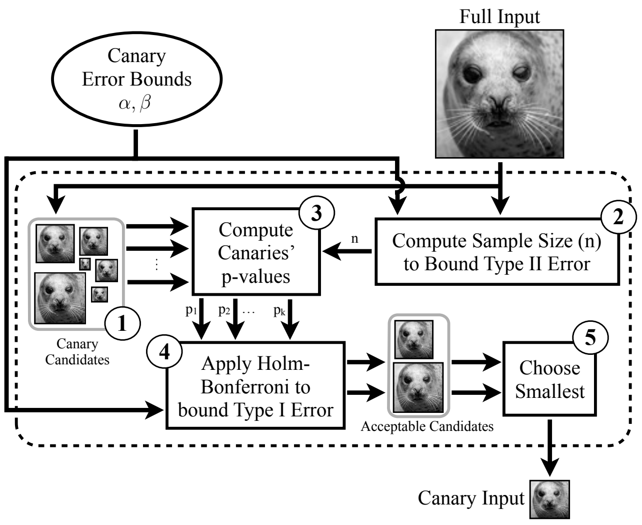

Full Input
Canary Error Bounds α, β
Compute Canaries' p-values
Compute Sample Size (n) to Bound Type II Error
n
p_1 p_2 … p_k
Canary Candidates
Apply Holm-Bonferroni to bound Type I Error
Acceptable Candidates
Choose Smallest
Canary Input
1
2
3
4
5

Figure 5. Canary input creation

4.1.2 Canary Candidates

The algorithm for creating a canary is illustrated in Figure 5. The inputs to the algorithm are the desired bound on the likelihood of getting a Type II error α, the desired bound on getting a Type I error β, and the full input to the program. The output of the algorithm is a small canary input deemed similar to the full input.

① **Generating Canary Candidates.** First, a set of candidate canaries $C_1, C_2, \ldots, C_k$ are generated. One of the main determinants a canary's quality is its size; a larger canary is likely to be a better reflection of the full input than a smaller canary. However, as the purpose of the canary is to use it in a dynamic search, a larger canary will also tend to result in a more expensive search. Our approach to generating candidates is to expose this inherent tradeoff, using candidates of many different sizes then choosing the smallest canary from among the candidates that is similar enough to the full input according to one of the metrics described in §4.1.4.

We generate $C_1, C_2, \ldots, C_k$ using regular, strided subsets of the full input. If N is the size of the full input, canary candidates are chosen that are size $^N/_{16}$, $^N/_{32}$, $^N/_{64}$, $^N/_{128}$ and $^N/_{256}$. We explicitly avoid selecting canaries larger than $^N/_{16}$, as canaries that are larger may take an unacceptably long time in the dynamic search, undermining the performance gains IRA aims to achieve by approximating the program. For one-dimensional inputs such as an array of scalars or an array of structs, an input of size $^1/_t$ is produced by taking every tth element from the input. For two-dimensional inputs such as matrices and images, an input of size $^1/_t$ is produced by taking every $^1/_{\sqrt{t}}$th element along both dimensions. This approach can easily be extended to higher-dimension inputs, however implementing that extension was not necessary for any of the programs used in this work.

	Mean	**Variance**	**Local Homogeneity**	**Autocorrelation**
Description	Mean μ of input elements	Variance σ^2 of input elements	Proportion Λ of elements represented by λ in canary $\notin [\lambda - \sigma z_{1-\alpha/2}, \lambda + \sigma z_{1-\alpha/2}]$	Correlation ρ among pairs of input elements (y_j, y_{j+1})
Null Hypothesis (H_0) **Alt. Hypothesis (H_A)**	$H_0 : \overline{\mu_i} = \mu_0$ $H_A : \overline{\mu_i} \neq \mu_0$	$H_0 : \overline{\sigma_i}^2 = \sigma_0^2$ $H_A : \overline{\sigma_i}^2 \neq \sigma_0^2$	$H_0 : \overline{\Lambda_i} \leq 0.1$ $H_A : \overline{\Lambda_i} > 0.1$	$H_0 : \overline{\rho_i} = \rho_0$ $H_A : \overline{\rho_i} \neq \rho_0$
Test Statistic (t_i)	$t_i = \frac{\mu_0 - \overline{\mu_i}}{\overline{\sigma_i}\sqrt{n}}$	$t_i = \frac{\overline{\sigma_i}^2}{\sigma_0^2}$	$t_i = \frac{\sqrt{n}\,\lvert\overline{\Lambda_i} - 0.1\rvert}{\sqrt{0.1(1-0.1)}}$	$t_i = \frac{ln\left(\frac{(1+\rho_0)(1-\overline{\rho_i})}{(1-\rho_0)(1+\overline{\rho_i})}\right)}{2\sqrt{n-3}}$
p-value (p_i)	$p_i = 2P(Z > t_i)$	$p_i = 2P(F_{n-1,n-1} > t_i)$	$p_i = 2P(Z > t_i)$	$p_i = 2P(Z > t_i)$
Sample Size (n)	$n = 2(z_{1-\alpha/2k} + z_{1-\beta/k})^2$	Formula yields no simple form; see Cohen [14] for details.	$g(x) = \sqrt{x(1-x)}$ $n = 0.1^{-2}(g(0.1)z_{1-\alpha/2k} + g(\overline{\Lambda_i})z_{1-\beta/k})^2$	$n = \frac{4(z_{1-\alpha/2k}+z_{1-\beta/k})^2}{ln((1+\rho_0)/(1-\rho_0))^2}$
Acceptability Test	*Holm-Bonferroni method:* sort p-values $p_1, p_2, \ldots p_k$ to obtain sorted p-values $p_{(1)}, p_{(2)}, \ldots p_{(k)}$. Find the minimum index m such that $p_{(m)} > \frac{\alpha}{k+1-m}$, then reject all canaries $C_{(i)}$ where $i \geq m$.			
Definitions	α: the desired bound on the probability of committing any Type I errors (false negative), β: the desired bound on Type II errors (false positive) k: the number of canary candidates, C_i: the ith canary candidate, $\overline{x_i}$: the sample statistic x for canary C_i, x_0: the sample statistic x for the full input Z: the standard normal distribution, z_y: the quantile function at y of Z, $F_{b,c}$: the F-distribution with degrees of freedom b and c			

Table 1. Similarity metrics used to assess canary similarity to full input, along with the relevant statistical formulas

4.1.3 Canary Selection

The remainder of the steps in this algorithm are focused on choosing the smallest canary from among these candidates that is similar to the full input.

② **Sample Size.** We next calculate the number of samples to take from each canary when evaluating their similarity. This calculation is designed to bound the likelihood of getting a Type II error when comparing those properties to the full input, discussed in further detail in §4.1.5. This sample size is denoted n.

③ **Canary Statistics.** We calculate the statistics needed to perform hypothesis tests on the canaries, taking a random sample of size n from each canary C_i, then use the samples to compute a test statistic for the canary t_i and a p-value p_i associated with that test statistic. We discuss tests statistics and p-values in more detail in §4.1.5, however the intuition is that t_i is simply a statistical measurement of the similarity between the canary and full input, while p_i is the statistical significance of that measurement.

④ **Canary Acceptability.** On the resulting p-values $p_1, p_2, \ldots, p_k$, we use the Holm-Bonferroni method, a technique designed specifically to bound the likelihood of getting a Type I error when performing multiple hypothesis tests [29], to partition the candidate canaries into two groups – those that are suitable representations of the full input because they are statistically similar enough to it, and those that are not.

⑤ **Select Canary.** Finally, the smallest of the suitable canaries is returned and used by IRA to perform a dynamic search for the most effective approximation. If no such canary is available, IRA immediately ceases approximation and begins to execute the exact version of the program.

4.1.4 Input Similarity Metrics

The purpose of the canary is to drive a dynamic search to determine how the program and its full input should be approximated. As such, it is of critical importance that the canary be similar to the full input. However, similarity can be measured in many ways. In this work, we consider four distinct definitions of similarity.

Mean. IRA supports using the arithmetic mean of the values in the canary and full inputs as the similarity metric. We define the mean of an input Y composed of values $y_1, y_2, \ldots, y_N$ as μ_Y. For convenience, the formal definition of μ_Y is supplied in Equation 1. A canary found to be acceptable according to this metric has an average value close to the average value of the full input.

$$\mu_Y = \frac{1}{N}\sum_{j=1}^{N} y_j \tag{1}$$

Variance. IRA also supports using the variance of values in the input as the similarity metric. The variance of Y is σ_Y^2, the definition of which is supplied in Equation 2. A canary that meets this standard of closeness will contain values that are dispersed to a degree similar to the dispersion found in the full input.

$$\sigma_Y^2 = \frac{1}{N}\sum_{j=1}^{N}(y_j - \mu_Y)^2 \tag{2}$$

Local Homogeneity. The canary is produced using a subset of the values in the full input. Thus, in essence, a single value in the canary embodies a (potentially large) number of values from the full input. To ensure the values in the canary are not highly dissimilar to the values in the full input they are supposed to embody, IRA leverages a measure of this dissimilarity. We denote this metric Λ_Y, defined by comparing each value y_j in the full input to λ_j, its representative value in the canary, and calculating the proportion of those values that are at least $z_{1-\alpha/2}$ standard deviations (see Table 1 for the definition of z) away from λ. The formal definition of Λ_Y is shown in Equation 3.

$$\Lambda_Y = \frac{1}{N}\sum_{j=1}^{N}\begin{cases}0 & \text{if } |y_j - \lambda_j| \leq \sigma_Y z_{1-\alpha/2} \\ 1 & \text{otherwise}\end{cases} \tag{3}$$

Autocorrelation. Last, IRA support measuring similarity between a canary and the full input by testing that their autocorrelations are similar. Autocorrelation is a special case of correlation, and is a measure of how similar each value in the input is to its neighbor. High coefficients of autocorrelation (those close to 1) indicate that neighboring values share a linear relationship across the input, while low coefficients (those close to zero) indicate no such relationship. Thus, autocorrelation detects small-scale patterns in the input. For an input Y, the coefficient of autocorrelation is ρ_Y. We provide a formal definition of autocorrelation in Equations 4 and 5.

$$Y' = \{y_1, y_2, \ldots, y_{N-1}\}, Y'' = \{y_2, y_3, \ldots, y_N\} \tag{4}$$

$$\rho_Y = \frac{1}{\sigma_{Y'}\sigma_{Y''}}\sum_{j=1}^{N-1}(y_j - \mu_{Y'})(y_{j+1} - \mu_{Y''}) \tag{5}$$

4.1.5 Statistical Underpinnings

At its core, canary selection in IRA is built on the statistical foundations of hypothesis testing, in which the evidential basis for hypotheses can be weighed statistically, allowing rejection of hypotheses that are not supported by the available evidence. For our purposes, the hypotheses considered are statements such as *canary C_i has the same autocorrelation as the full input*. Such a hypothesis can be rejected if a comparison between the full and canary inputs does not provide sufficient evidence to support the hypothesis. Thus, by rejecting the hypothesis equating the canary to the full input we reject the canary. Alternatively, when the hypothesis test fails to reject the null hypothesis, the canary is deemed to be acceptably similar to the full input.

It is important to note that IRA may need to consider many such hypotheses when constructing a canary, and thus is subject to the *multiple testing problem* [37]. The multiple testing problem describes a problem wherein evaluating the validity of multiple hypotheses can considerably increase the likelihood of incorrectly evaluating at least one of the hypotheses. Consider a hypothesis concerning the fairness of a coin, where we wish to assess the validity of the hypothesis by flipping the coin 10 times and calling it biased if we flip at least 9 heads or tails. Applying this test to 1 unbiased coin, it is unlikely that it will appear unfair, a probability of 2.1%. However, if we apply this test to 100 coins, there is a very strong likelihood (88.6%) that at least one coin will be judged to be unfair, an incorrect determination. Similarly, the multiple testing problem applies to our hypotheses about canaries. Therefore, when evaluating canaries we adjust our statistical methods by incorporating the Bonferroni correction for Type I errors and the Holm-Bonferroni method for Type II errors to ensure that we avoid the multiple testing problem. These adjustments are detailed shortly.

Hypothesis Testing. In a hypothesis test, we propose two hypotheses relating to the similarity of a canary to the full input. These hypotheses are called the null hypothesis H_0 and the alternative hypothesis H_A. For each canary C_i, we construct null and alternative hypotheses and determine whether to accept or reject C_i based on the evidence found in favor of the null hypothesis. Our discussion will focus on hypothesis testing for the arithmetic mean of the input, however IRA supports several other metrics that have been discussed previously and are summarized in Table 1. These other metrics can be used by substituting their equations in place of the equations described for the mean.

A hypothesis test for the mean takes the form shown in Equation 6, where $\overline{\mu_i}$ is the sample mean of canary C_i and μ_0 is the sample mean of the full input.

$$\begin{aligned} H_0 &: \overline{\mu}_i = \mu_0 \\ H_A &: \overline{\mu}_i \neq \mu_0 \end{aligned} \tag{6}$$

Next, the truth of H_0 is evaluated by calculating and evaluating a test statistic. The test statistic is used to produce a p-value for the test, the probability of attaining a test statistic at least as extreme as the observed test statistic given that the null hypothesis is true. Thus, the smaller the p-value, the lower the probability of the observed test statistic appearing if the null hypothesis is true. Some significance level α is chosen as a cutoff point for the hypothesis test, where $p \leq \alpha$ causes the null hypothesis to be rejected. In particular, to compute the t-statistic and p-value for the mean, we use the standard formulas shown in Equations 7 and 8.

$$t_i = \frac{\mu_0 - \overline{\mu_i}}{\overline{\sigma_i}\sqrt{n}} \tag{7}$$

$$p_i = 2P[Z > t_i] \tag{8}$$

Standard single-comparison hypothesis tests stop here, rejecting the null hypothesis if $p_i \leq \alpha$. However, we must take further steps to avoid the multiple comparisons problem.

Controlling Type I Errors. The Holm-Bonferroni method considers multiple hypotheses simultaneously [29]. It outputs a set of hypotheses that are rejected, and a set that are not, where the probability of obtaining *any* Type I errors is bound by α. The method begins by sorting the p-values $p_1, p_2, \ldots, p_k$ from lowest to highest, resulting in a new indexing of p-values $p_{(1)}, p_{(2)}, \ldots p_{(k)}$ corresponding to null hypotheses $H_{(1)}, H_{(2)}, \ldots H_{(k)}$. It then rejects hypotheses $H_{(1)}, H_{(2)}, \ldots H_{(m-1)}$, where m is the minimum index satisfying Equation 9.

$$p_{(m)} > \frac{\alpha}{k+1-m} \tag{9}$$

The result of this method is a set of null hypotheses $H_{(m)}$, $H_{(m+1)}$, ..., $H_{(k)}$ that are not rejected, corresponding to a set of canaries $C_{(m)}, C_{(m+1)}, \ldots C_{(k)}$ that are deemed to be suitably similar to the full input.

Controlling Type II Errors. Given desired bounds α and β on the likelihood of getting any Type I or Type II errors, respectively, the standard formula for computing the number of samples needed to ensure the likelihood of getting a Type II error of no more than β in a single comparison hypothesis test is shown in Equation 10.

$$n = 2(z_{1-\alpha/2} + z_{1-\beta})^2 \tag{10}$$

To account for the multiple testing problem when using k canaries, we use the Bonferroni correction [16], substituting α/k and β/k in place of α and β in Equation 10.

$$n = 2(z_{1-\alpha/2k} + z_{1-\beta/k})^2 \tag{11}$$

This adjusted formula requires an increased sample size over the non-adjusted formula. However, sampling overhead remains reasonable even for large numbers of canary candidates (large k) because the sample size due to this adjustment grows sublinearly as k increases [60].

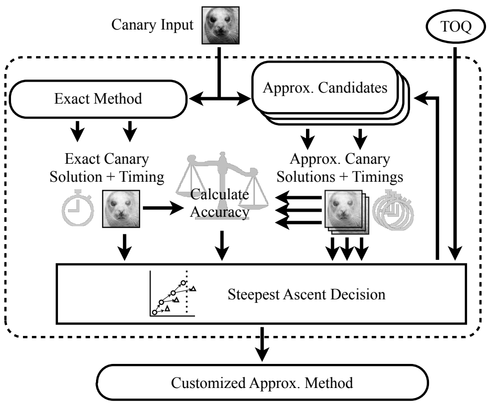
Smallest Acceptable Candidate. All of the canary candidates that remain from the preceding set of steps are suitably similar to the full input. However, it is important that we choose the acceptable candidate that results in the shortest search time in IRA's next step. Thus, the canary construction algorithm terminates by choosing the smallest from among the remaining acceptable candidates.

4.2 Choosing an Effective Approximation

IRA uses the canary input to rapidly and dynamically choose how to approximate the program on the full input. This section describes how that choice is made.

4.2.1 Definition of Result Accuracy

Controlling and maintaining sufficiently accurate computation is important in approximate computing [49, 50, 56]. Prior work has pointed out that result accuracy is domain, application, and context dependent [38] and includes such varied metrics as the scaled difference between output, the peak signal to noise ratio (PSNR) or the average absolute output accuracy. Therefore, we design IRA to be agnostic to the specific method used to calculate result accuracy. That is, we assume only that the application developer provides a well-defined accuracy calculation function. Our formulation of this function $F_{accuracy}$, given two solutions S_{exact} and S_{approx}, computes a single accuracy metric $\delta \in [0, 1]$ describing the accuracy of S_{approx} relative to S_{exact}. $F_{accuracy}$ is leveraged by IRA to compute the accuracy of a number of approximations on the canary input, comparing them to the solution produced by the exact method on the canary input. We assume also that the user of



Canary Input
TOQ
Exact Method
Approx. Candidates
Exact Canary Solution + Timing
Calculate Accuracy
Approx. Canary Solutions + Timings
Steepest Ascent Decision
Customized Approx. Method

Figure 6. Search for approximation using canary

the application supplies a minimum acceptable result accuracy, called the target output quality (TOQ).

4.2.2 Where and How to Approximate

There may be a number of code regions amenable to approximation in an program. Consider an program with two disjoint loops that can be approximated with loop perforation [27, 45] and tiling [50], respectively. In IRA, each such code region is an approximation opportunity, and IRA treats each approximation opportunity as one dimension in a multi-dimensional search space by encoding the parameters for each approximation opportunity as one of a range of integral values $\{1, \ldots, v\}$. In the encoding, the value 1 has special meaning, and is used to represent the exact computation in lieu of approximation.

Many approximation mechanisms can be parameterized, such as the rate at which iterations are skipped in loop perforation or the size of one side of a tile in tiling approximation. In such cases, numbers larger than 1 encode each value that can be taken by a parameter. Our search algorithm makes only the assumption that larger values correspond to more aggressive approximation (i.e., that it runs faster but has lower accuracy). By encoding the search space in this fashion, IRA has the option to select the exact computation at each approximation opportunity, allowing it to choose where to approximate. By selecting between the values larger than 1, IRA determines how aggressively to take advantage of each approximation opportunity.

4.2.3 Search for an Effective Approximation

IRA uses a greedy approach based on steepest ascent hill climbing [48] to tune the parameters for the available approximations and choose *how to approximate*, using the approach presented in Figure 6. Each possible choice of how to approximate is defined as a point in an m-dimensional space $(d_1, d_2, \ \ldots \ , d_m)$, where each dimension is an encoded range of integral values as described previously. IRA first evaluates the point $(1, 1, \ \ldots \ 1)$ on the canary, which is the exact solution to the program on the canary. This so-

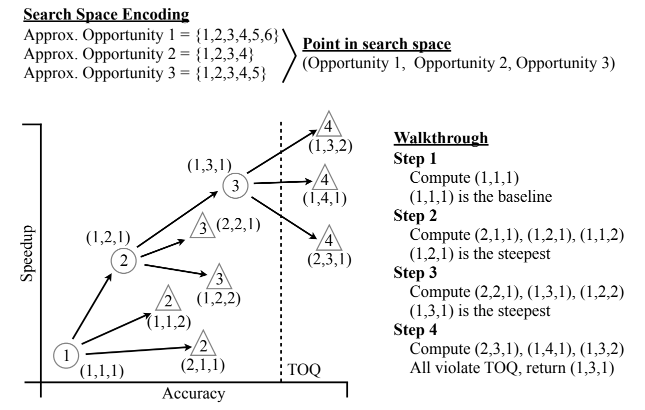

Search Space Encoding
Approx. Opportunity 1 = {1,2,3,4,5,6}
Approx. Opportunity 2 = {1,2,3,4}
Approx. Opportunity 3 = {1,2,3,4,5}
Point in search space
(Opportunity 1, Opportunity 2, Opportunity 3)
Speedup
Accuracy
TOQ
(1,1,1)
(1,2,1)
(1,1,2)
(2,1,1)
(1,3,1)
(2,2,1)
(1,2,2)
(1,3,2)
(1,4,1)
(2,3,1)
Walkthrough
Step 1
Compute (1,1,1)
(1,1,1) is the baseline
Step 2
Compute (2,1,1), (1,2,1), (1,1,2)
(1,2,1) is the steepest
Step 3
Compute (2,2,1), (1,3,1), (1,2,2)
(1,3,1) is the steepest
Step 4
Compute (2,3,1), (1,4,1), (1,3,2)
All violate TOQ, return (1,3,1)

Figure 7. Example search for an effective approximation

lution is used as a timing and accuracy baseline, against which approximate solutions are evaluated. Beginning at $(1, 1, \ \dots \ 1)$, IRA then iteratively evaluates the incrementally more aggressive value for each of the tuning parameters, computing the accuracy and speedup relative to the exact version, then selects the increment that both satisfies the TOQ set forth by the user and yields the steepest slope in terms of accuracy vs. speedup. If no such increment exists, the search terminates. If such an increment exists, it is used as the starting point for the next iteration. Upon termination, the last valid point is returned by the search and is used to approximate the full program input.

In executing the search algorithm, IRA runs the exact computation in addition to a number of approximation alternatives on the canary. The execution of this search is typically very fast relative to exact execution on the full input because the amount of computation needed in regularly-structured computation depends substantially on the size of the canary, which is much smaller than the full input. We quantitatively evaluate the time spent in the search in §5, showing that it equates to an average of 3.2% of exact execution time on a suite of test programs.

Example Search. Consider Figure 7, which shows an example search over 3 approximation opportunities. In step 1 IRA evaluates $(1, 1, 1)$ as a baseline. In step 2, the next available increment along each dimension is tested – $(2, 1, 1)$, $(1, 2, 1)$ and $(1, 1, 2)$ in this case. $(1, 2, 1)$ is found to have the steepest ascent in the speedup/accuracy space, and is used as the baseline for the next step. In step 3, $(2, 2, 1)$, $(1, 3, 1)$ and $(1, 2, 2)$ are tested, and $(1, 3, 1)$ is found to have the steepest ascent. Finally, in step 4 $(2, 3, 1)$ $(1, 4, 1)$ and $(1, 3, 2)$ are tested. Each of the configurations is found to violate the TOQ bound and the search terminates. IRA returns $(1, 3, 1)$ as the most effective approximation.

4.3 Putting it all Together

Final Approximation. Once IRA has chosen an effective approximation for the canary, that chosen approximation is used to run the program on the full input to produce a final, approximate result.

Runtime Safety. In approximate computing, altering computation to trade performance for accuracy, particularly when discarding computation, can have the effect of changing control flow, producing unsafe intermediate results (e.g., a 0 that will be used as the denominator in a division operation), or memory accesses that corrupt state or result in access violations, resulting in runtime faults that were not anticipated by the application programmer. Prior work has shown that it is often possible to recover from memory errors using checkpointing [55] or heap replication [8], and from floating point errors using reevaluation or rollback [24], resuming computation to successfully produce a result. We have experienced no such faults in our experiments, however if necessary IRA can be augmented to include mechanisms to guard against such faults.

5. Evaluation

We thoroughly evaluate IRA to examine its impact on performance and result accuracy.

5.1 Methodology

Applications and Inputs. We evaluate 13 applications that use between 2 and 800 inputs. These applications cover a number of important problem domains that include image processing, data mining, machine learning and computer vision. Applications and inputs are summarized in Table 2.

Approximation Techniques. Our experiments bring input responsiveness to four classes of approximate computing techniques. The approximation techniques themselves have limitations in terms of how they can be applied. For example, tiling approximation requires iterative computation on image pixels, and thus is applied only to CrossCorr, Gamma and GaussianFilter, while numerical approximation requires particular mathematical constructs to be present in the program (e.g., the use of trigonometric functions in Inversek2j). In many cases, multiple approximations are used side-by-side among an application's different code regions. A summary of which approximations are applied to which benchmarks is summarized in Table 2. Short code samples illustrating each of the approximations are given in Table 3. The approximations used in the evaluation are as follows:

- **Loop Perforation [27, 45] –** loop perforation discards iterations in a loop. We use either unadjusted perforation, where every nth iteration in a loop is executed, or extrapolated perforation, which is similar to unadjusted perforation but extrapolates computed results to make up for the skipped iterations. Loop perforation can be made more aggressive by using larger values of n. Loop perforation is used in CrossCorr, FuzzyKmeans, LucasKanade, Kernel, Kmeans, MatMult, MeanShift and ScalarProd.

Application	Description	Domains	Input Suite	Approximation(s) Used
CrossCorr	Measure signal/image similarity over sliding window	Pattern recognition, cryptanalysis, neurophysiology	800 IMAGE	4× perforate, 1× 2D-tile
FuzzyKmeans	Cluster with fuzzy cluster membership	Machine learning, data mining	4 SVM	5× perforate
Gamma	Apply gamma correction to an image	Image processing	800 IMAGE	1× 2D-tile
GaussianFilter	Apply a Gaussian filter to an image	Image processing	800 IMAGE	1× 2D-tile
Integration	Numeric integration of transcendentals	Scientific computing, engineering	19 EQN	1× numerical approx.
Inversek2j	Kinematics for 2-joint arm	Robotics	90 ANGLE	4× numerical approx.
Jmeint	Triangle intersection detection	3D gaming	40 TRI	1× algorithm choice
LucasKanade	Optical flow estimation	Computer vision	2 PERFECT	2× perforate
Kernel	Estimate a probability density function	Machine learning, signal processing, econometrics	2 KDDCUP	4× perforate
Kmeans	Cluster points for classification	Mach. learning, data mining	4 SVM	4× perforate
MatMult	Matrix-matrix multiply	Machine learning, scientific computing, game theory	40 PDF	2× perforate
MeanShift	Apply mean shift to an image	Computer vision, image processing	4 SVM	3× perforate
ScalarProd	Dot product of two vectors	Mechanics, machine learning, graphics	40 PDF	2× perforate

Input Suite	Description
90 ANGLE	Sets of angles drawn from 90 different probability distributions
19 EQN	Sets of equations containing polynomials with different max degree
800 IMAGE	A database of 800 images
2 KDDCUP	1999 KDD Cup data set from the UCI Machine Learning Repository [5]
40 PDF	Probability dists used: beta, binomial, cauchy, chi-squared, exponential, f, gamma, geometric, hyper, log-normal, normal, poisson, t, uniform, weibull
2 PERFECT	Medium and large inputs from the PERFECT benchmarks [7]
4 SVM	Support vector machines from the UCI Machine Learning Repository [5]
40 TRI	Sets of triangles in the unit cube, varying distributions of triangle sizes

Table 2. Applications and input sets used in the evaluation

	Loop Perforation	Tiling	Algorithm Choice	Numerical Approximation

Exact Code Sample	`for (i=0; i<N; i++)`
	`sum += A[i];`
		`for (i=0; i<N; i++)`
		`B[i] = f(A[i]);`
			`y = f_impl1(x);`	`y = cos(x);`

Approximate Code Sample	`for (i=0; i<N; i+=n)`
	`sum += n*A[i];`
		`for (i=0; i<N; i+=n){`
		`B[i] = f(A[i]);`
		`for (j=1; j<n; j++)`
		`B[i+j] = B[i]; }`
			`y = f_impl2(x);`	`y = (x - x*x/2.0);`

Table 3. Examples of the approximations used in the evaluation; code impacted by approximation is highlighted

- **Tiling [50] –** instead of computing each element of an output, tiling computes a single output element and projects it onto the surrounding elements to form a tile. Tiling approximation is made more aggressive by using larger tile sizes. We use tiling approximations in CrossCorr, Gamma and GaussianFilter.
- **Algorithm Choice [3, 15] –** we use IRA to choose between five different algorithmic implementations of Jmeint that offer different accuracy-performance tradeoffs in computing whether pairs of 3D triangles intersect. The most complex algorithm is the exact algorithm, while the simplest algorithm uses computationally cheap heuristics that work well only when triangles are far apart.
- **Numerical Approximation [25] –** we use numerical approximation techniques within Integration and Inversek2j. Integration numerically integrates a non-integrable set of equations using the trapezoid method, which can be made faster and less accurate by using fewer trapezoids. Inversek2j involves a motion calculation that relies on the trigonometric functions $sin(x)$, $cos(x)$, $sin^{-1}(x)$, $cos^{-1}(x)$. We approximate these trigonometric functions by using the first one (sin and sin^{-1}) or two (cos and cos^{-1}) terms of the function's Taylor series in lieu of the precise library implementation. These approximations are accurate when x is near zero, and become less accurate farther away from zero. Thus, we can trade speed for accuracy by choosing a k such that approximation is used only when $|x| < k$, making the approximation more aggressive by using larger values of k.

Platform. All results are collected on a stock 2.4GHz Intel Xeon E5-2407v2 (Ivy Bridge) server running Linux kernel 3.11.0. Applications are executed on the server in serial, and task pinning is used to prevent migration.

Error Bounds, TOQ and Accuracy. All experiments in this evaluation use canary error bounds $\alpha = \beta = 0.05$, thus obtaining Type I and Type II error bounds of 0.05. TOQ values ranging from 90% to 97.5% are used in the evaluation, specified for each experiment. IRA is agnostic to the accuracy metric, simply using the supplied definition of accuracy (see §4.2.1) and configuring the approximation so as to not violate the accuracy target set forth by the user. In our evaluation, we use miss rate as the accuracy metric for Jmeint, absolute relative error for ScalarProd and average centroid distance from the origin in Kmeans and FuzzyKmeans. For all other applications, accuracy is defined as the average of element-wise absolute relative error.

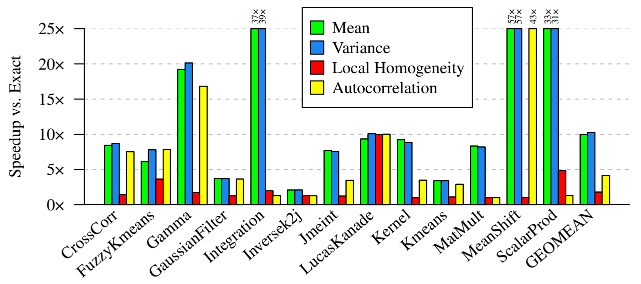


Figure 8. Comparison of canary similarity metrics

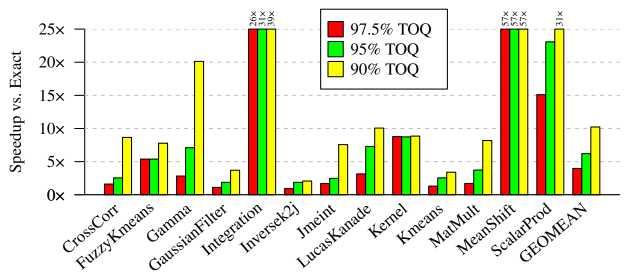


Figure 10. Speedup of IRA across three TOQs

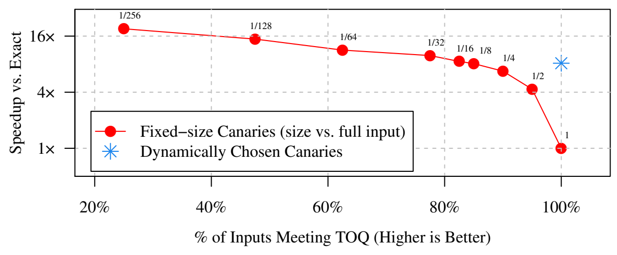


Figure 9. Speedup and number of TOQ violations for dynamically chosen canaries (blue star) vs. fixed-size canaries (red circles) on MatMult; all fixed size canaries achieve lower speedup, more TOQ violations, or both

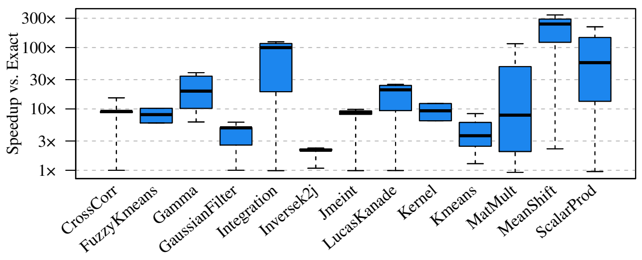


Figure 11. Distribution of speedups across inputs for IRA at 90% TOQ, illustrating the wide range of approximations dynamically chosen across different inputs; larger speedups occur when more aggressive approximation is applied

5.2 Canary Construction

Similarity Metrics. Figure 8 presents the speedup over exact computation obtained by IRA when employing each of the four canary similarity metrics described in §4.1 at a TOQ of 90%. As the figure illustrates, the largest speedups obtained are for variance and mean, averaging 10.2× and 9.9×, respectively. The speedups obtained when using autocorrelation are modest, averaging 4.3×, while using local homogeneity causes a speedup of 2.1×.

This is a non-intuitive result, as the simpler metrics – Mean and Variance – perform better while achieving similarly low counts of TOQ violations (TOQ is violated on less than 1% of inputs on average for all metrics). Closer inspection reveals that autocorrelation and local homogeneity are more difficult similarity metrics to satisfy, thus they often result in either (1) choosing larger canaries, leading to longer search times, which diminishes the overall speedup or (2) finding no acceptable canaries, and thus no approximation being used. This is particularly true for local homogeneity, which achieves speedups near 1 for 8 of the 13 applications. This leads us to the insight that more complicated is not always better; autocorrelation and local homogeneity reject many canaries that are deemed acceptable according to their mean and variance – canaries that turned out to be perfectly adequate in searching for an effective approximation.

A second insight revealed by this data is that mean and variance do not significantly differ from one another in terms of the canaries selected, a fact that holds true on average and among the individual applications. This suggests that both metrics produce reasonable canaries and function well across a range of problems and domains. Because variance is a slight improvement over mean in terms of the overall speedup of IRA, the remainder of the experiments use variance as the similarity metric when constructing canaries.

Sizing Canaries Dynamically. Dynamically-sized canaries are valuable because they yield approximations that are just aggressive enough for each input. This means that (1) each canary is no larger than necessary for well-behaved inputs, thus keeping the overhead low during the approximation search process, and (2) each canary can be made large enough to adequately represent the full input.

To illustrate the value of dynamic canary creation, we compare the results of using IRA's dynamically-chosen canaries to canaries created using a range of fixed strategies when approximating MatMult to at a TOQ of 90%. The inputs to MatMult are the set of 40 inputs described in Table 2, spanning a range of probability distributions that include long tail and high variance distributions. The results are illustrated in Figure 9, which shows the speedup (y-axis) and number of inputs meeting TOQ (x-axis) achieved by IRA when using fixed-size canaries (red circles and line). As the figure shows, there is a tradeoff between speedup and input violations to be made when using fixed-size canaries: smaller canaries produce larger speedups but large numbers

Application	Meets TOQ	% of Inputs
CrossCorr	790 / 800	98.8%
FuzzyKmeans	4/ 4/	100%
Gamma	752 / 800	94.0%
GaussianFilter	797 / 800	99.7%
Integration	19 / 19	100%
Inversek2j	90 / 90	100%
Jmeint	40 / 40	100%
LucasKanade	4 / 4	100%
Kernel	2 / 2	100%
Kmeans	4 / 4	100%
MatMult	40 / 40	100%
MeanShift	4 / 4	100%
ScalarProd	40 / 40	100%
MEAN	-	**99.4%**

Table 4. The proportion of inputs for which IRA hits the target output quality (TOQ) at TOQ=90%; there are no TOQ violations in 10 of the 13 applications

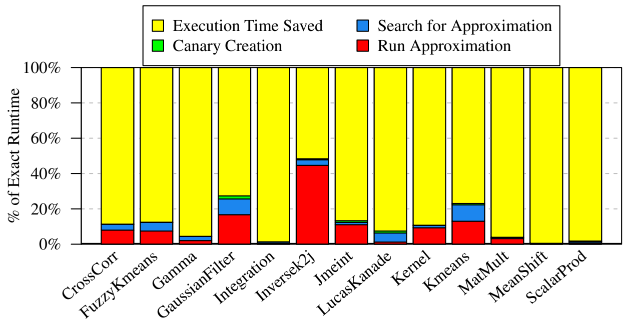


Figure 12. Breakdown of time spent by IRA, showing time to create the canary (barely visible), choose the approximation, and run the chosen approximation on the full input

of TOQ violations, while larger canaries produce fewer TOQ violations but smaller speedups. Improving in both dimensions is the point illustrating the speedup (8.2×) and TOQ violations (0%) achieved when using dynamically-chosen canaries (blue star). This demonstrates the advantage of using dynamically-chosen canaries – a small canary is used if a small canary can serve as a suitable representation of the full input, while a large canary is used if not.

5.3 IRA Speedup and Accuracy

Speedup. We refer next to Figure 10, which presents the average speedup achieved by IRA relative to the runtime of the exact computation across three TOQ values: 97.5%, 95% and 90%. Each application is run on all inputs, and the speedups presented are the geometric mean of speedup across the inputs. Performance measurements of IRA are the end-to-end runtime, including the time to produce the canary input, search for the customized approximation and run that approximation on the full input. As one would expect, IRA achieves speedups that scale up as the TOQ is relaxed, ranging from an average of 3.9× at 97.5% TOQ up to 10.2× at 90% TOQ.

Dynamism. Figure 11 presents boxplots of IRA speedups across inputs for each application at TOQ=90%. The boxplots highlight the maximum (upper whisker), 75th percentile (box upper edge), median (line within box), 25th percentile (box lower edge) and minimum (lower whisker) speedups. The large range of speedups shown in Figure 11 highlights the key feature of IRA: different inputs to the same application can be more or less difficult to approximate. IRA takes advantage of these differences to choose the right approximation for each input and maximize performance when applying approximation. This degree of dynamism allows IRA to realize significantly higher performance for many cases that cannot be taken advantage of by conventional approaches that apply one approximation across inputs. We discuss this in greater detail in §5.5, comparing IRA to oracle versions of two classes of prior work that apply the same approximation across multiple inputs.

Accuracy. The accuracy of the results produced by IRA is presented in Table 4, showing the number of TOQ violations across inputs at TOQ=90%. On average, IRA meets TOQ for over 99% of inputs. Furthermore, for 10 of 13 applications there are no output quality violations, and the maximum proportion of TOQ violations is 6% for Gamma. Moreover, those cases that violate TOQ are typically not far from TOQ. For instance, 78% of violating cases have an output quality of 88% or better (within 2% of TOQ). From this we conclude that IRA works very well at producing a minimal number of TOQ violations in practice, however we take care to note that IRA makes no guarantees about output accuracy.

5.4 Overhead Analysis

Figure 12 presents a breakdown of the time spent by IRA in various stages of execution as a fraction of the total runtime of the exact application run (TOQ=90%). These percentages are the average across all inputs for each application. The bulk of the time shown in the figure is execution time saved by approximating the application with IRA (yellow). We divide the execution time of IRA into three parts: the time spent creating a canary input (green; barely visible), the time spent using the canary to search for an approximation (blue), and the time spent running the chosen approximation on the full input (red).

The time spent choosing the canary is small, which is to say that the remaining bottlenecks in IRA are elsewhere. Many applications – Gamma, Integration, Inversek2j, Jmeint, Kernel, MatMult, MeanShift and ScalarProd – spend a small proportion of the time searching for the the approximation, while many others spend a more appreciable fraction of time in the search. Large search times are caused by a combination of large canaries and high-dimensional search spaces (that is, those that have a larger number of approximation opportunities to explore).

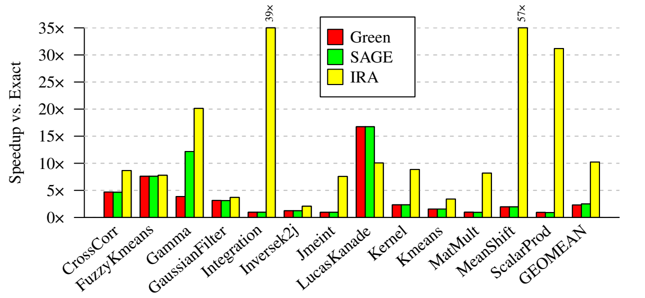


Figure 13. Comparison of IRA to calibration-based approximation with Green [6], SAGE [49], showing that IRA achieves more than 4× the speedup of both

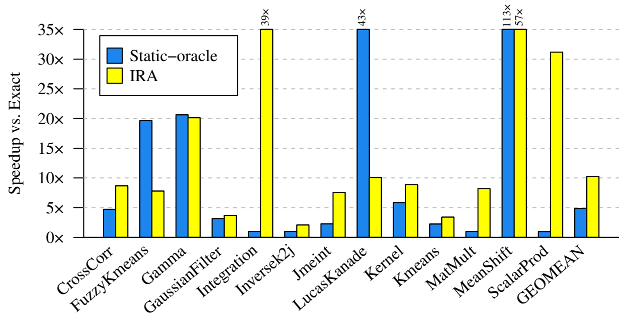


Figure 14. Comparison of IRA to a static oracle, showing that IRA achieves more than 2× the speedup

The size of the approximation search spaces varies significantly across applications, ranging from 5 in the case of Jmeint (5 versions of the algorithm constitute the search space) to 22,500 in the case of CrossCorr where 5 approximation options are parameterized. Our hill climbing algorithm takes $\mathcal{O}(m * n)$ steps, where m is the number of approximations to parameterize and n is the number of ways to parameterize a single approximation. In our experimentation, we have found that searches often end in fewer than 10 steps and typically take no more than a few dozen steps, ultimately resulting in searches that average 3.2% of exact execution time.

The search time for choosing the approximation might be reduced by paring down the number of approximation opportunities and parameter ranges to reduce the size of the search space. For example, if certain approximation opportunities were revealed through static analysis, offline profiling, or feedback from earlier runs of IRA to result in ineffective approximations for a substantial fraction of inputs, those opportunities could be discarded. However, because the goal in this work is to automate the process of choosing the approximation without the aid of offline profiling or analysis, we implement no such mechanism.

5.5 Comparison to Prior Work

Oracle Calibration Approximation. Green and SAGE are two state-of-the-art calibration systems that dynamically tune approximation to control TOQ violations [6, 49]. Green uses profiling in concert with calibration at fixed intervals (e.g., every 10 inputs) to tune how aggressively to apply approximation. SAGE is also calibration-based, however it is entirely dynamic in nature and it continually changes the calibration period as more inputs are seen, lengthening the period when calibration shows that the current tuning of the approximation does not violate TOQ.

We compare IRA to using oracle versions of the SAGE and Green runtime systems to choose approximations. The oracle versions of SAGE and Green runtimes do two things perfectly that the systems themselves cannot do in practice. First, calibration on an input yields the precise speedup and accuracy for that input on *all approximations*, allowing the approximation to be tuned to exactly the most effective approximation at each calibration point. Second, calibration intervals are given by oracle for each application. For Green, the best calibration interval is used out of all possible calibration intervals, and for SAGE both the best possible calibration interval and calibration adjustments are used. Thus, our experiments represent upper bounds for the speedups achievable on these applications, inputs and approximation techniques with Green and SAGE.

We compare IRA against the oracle Green and SAGE by holding the number of TOQ violations achieved by each approach constant and examining the speedups achieved. The results of this comparison are shown in Figure 13, which shows the speedup of the three techniques where the TOQ violations are held constant at the TOQ violations IRA achieves at 90% TOQ. IRA improves the performance by an average of 10.2× by customizing the approximation to each individual input, while oracle Green speeds up by an average of 2.2× and oracle SAGE speeds up by a factor of 2.3×.

There are a number of programs for which the oracle Green and SAGE provide no speedup, such as MatMult and ScalarProd. This occurs because some of the inputs have high variance or long tails, producing poor quality results when applying approximation. For these programs, Green and SAGE get locked into unnecessarily conservative approximation approaches for a series of inputs once calibration has been done on a difficult input. IRA, on the other hand, uses conservative approximations on these difficult inputs while applying appropriately aggressive approximation on others. On LucasKanade, Green and SAGE achieve more speedup than IRA. This occurs because LucasKanade can be aggressively approximated on all inputs, thus allowing Green and SAGE to calibrate once and run those aggressive approximations for all input, whereas IRA spends valuable time searching for an approximation on all inputs.

Oracle Static Approximation. We next compare IRA to a *static oracle* approach to choosing approximation. A static oracle approach to approximation is one that has full knowledge of all inputs, as well as how the approximation options fare in terms of performance and accuracy for those inputs. Such a static oracle is instructive for understanding the limits of profile-guided approaches. We derive a comparison between IRA and the static oracle by using the static oracle to choose one approximation per application from the approximation options described in Table 2 such that the average speedup is as large as possible without violating TOQ more than IRA. At TOQ=90%, we find that the static oracle achieves an average speedup of 4.9×, where IRA achieves a speedup of 10.2×. This result demonstrates that approaches rooted in choosing one approximation to cover numerous inputs serves as a significant limitation on the speedups achievable by static techniques.

6. Related Work

There are many approaches for trading result accuracy for decreased execution time or energy, based on some combination of programmers [10], runtime systems [6, 26, 58], programming languages [3, 51], middleware [1, 19], compilers [49], and hardware [2, 17, 17, 23, 33, 41, 52, 61].

Some approaches to software-based approximation use formal analysis to provide worst-case guarantees [11, 12, 39, 54], while others use calibration offline [3, 27, 38, 40] or at runtime [1, 49, 50] to guide approximation. Others have proposed software [22, 47] and hardware [30] systems to catch highly inaccurate approximations early in their execution. SAGE [49] uses a dynamic calibration interval coupled with steepest ascent decisions based on the result accuracy. Another body of related research analyzes the accuracy or robustness of programs in the event of faults [31, 32, 57] or uncertain input data [9, 53], which has been used to locate code regions to approximate or bound the accuracy of approximate computation [11–13, 38].

Approximation has been performed by decreasing the number of iterations or tasks executed [27, 34, 56] or by replacing exact operations with less accurate versions [40, 54]. One such replacement strategy is to relax synchronization in parallel architectures [43, 49, 59]. Misailovic et al. [40] replace loops with parallel loops. ApproxHadoop [19] leverages statistical techniques to provide accuracy guarantees when applying approximation to MapReduce applications. Branch and data herding [54] eliminate warp divergence in GPGPUs, selecting the most common branch or memory access for the entire warp.

Compilers and frameworks have been used to facilitate selecting between multiple programmer-supplied implementations [3, 15, 58, 62]. Loop perforation was used by Hoffmann et al. [27] and can incorporate extrapolation to correct bias in the result [45], similar to the work on task skipping by Rinard [44]. Discarding tasks is a similar method to loop perforation in principle, but items in a queue are discarded rather than iterations in a loop [46].

7. Conclusion

This work motivates and introduces *Input Responsive Approximation* (IRA), a novel approach for automatically choosing how to approximate programs on an input-by-input basis. IRA accomplishes this by producing a *canary input* at the program outset, a reduced version of the full input rigorously constructed to retain the properties of the full input. This canary is used to rapidly test and choose from among the available approximations. We use IRA to approximate 13 image processing, machine learning, data mining, and computer vision applications. Using these applications, we show that IRA achieves an average speedup of 10.2× at a target output quality of 10%, far higher than oracle versions of state-of-the-art prior work.

Acknowledgements

We would like to thank our anonymous reviewers for their comments and suggestions. We also thank our shepherd, Cindy Rubio González, for her valuable feedback. This work was supported by National Science Foundation grant XPS-1438996. This work was also partially supported by the U.S. Department of Energy's Office of Science, Office of Advanced Scientific Computing Research, under award DE-SC0014134.

References

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *European Conference on Computer Systems (EuroSys)*, 2013.

[2] R. S. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger. General-purpose code acceleration with limited-precision analog computation. In *International Symposium on Computer Architecture (ISCA)*, 2014.

[3] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. *Code Generation and Optimization (CGO)*, 2011.

[4] Apple. Siri. `https://www.apple.com/ios/siri/`, 2016. [Online; accessed 10-April-2016].

[5] K. Bache and M. Lichman. UCI machine learning repository. `http://archive.ics.uci.edu/ml/`, 2016. [Online; accessed 10-April-2016].

[6] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Programming Language Design and Implementation (PLDI)*, 2010.

[7] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and A. Tumeo. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute, 2013.

[8] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Programming Language Design and Implementation (PLDI)*, 2006.

[9] J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain<T>: A first-order type for uncertain data. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[10] S. Byna, J. Meng, A. Raghunathan, S. Chakradhar, and S. Cadambi. Best-effort semantic document search on GPUs. In *Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2010.

[11] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *Programming Language Design and Implementation (PLDI)*, 2012.

[12] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013.

[13] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving programs robust. In *Foundations of Software Engineering (FSE)*, 2011.

[14] J. Cohen. *Statistical power analysis for the behavioral sciences*. Lawrence Erlbaum, 1988.

[15] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *Programming Language Design and Implementation (PLDI)*, 2015.

[16] O. J. Dunn. Multiple comparisons among means. *Journal of the American Statistical Association*, 1961.

[17] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[18] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *International Symposium on Microarchitecture (MICRO)*, 2012.

[19] I. Goiri, R. Bianchini, S. NagaraKatte, and T. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[20] Google. Glass. `https://www.google.com/glass/start/`, 2016. [Online; accessed 10-April-2016].

[21] Google. Google Now. `http://www.google.com/landing/now/`, 2016. [Online; accessed 10-April-2016].

[22] B. Grigorian and G. Reinman. Dynamically adaptive and reliable approximate computing using light-weight error analysis. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2014.

[23] B. Grigorian, N. Farahpour, and G. Reinman. Brainiac: Bringing reliable accuracy into neurally-implemented approximate computing. In *High Performance Computer Architecture (HPCA)*, 2015.

[24] J. R. Hauser. Handling floating-point exceptions in numeric programs. *Transactions on Programming Languages and Systems (TOPLAS)*, 1996.

[25] F. B. Hildebrand. *Introduction to numerical analysis*. Courier Corporation, 1987.

[26] H. Hoffmann, J. Eastep, M. Santambrogio, J. Miller, and A. Agarwal. Application heartbeats for software performance and health. *MIT CSAIL Technical Report*, 2009.

[27] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. *MIT CSAIL Technical Report*, 2009.

[28] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[29] S. Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 1979.

[30] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. Rumba: an online quality management system for approximate computing. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[31] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra. ERSA: Error resilient system architecture for probabilistic applications. In *Design, Automation and Test in Europe (DATE)*, 2010.

[32] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. SWAT: an error resilient system. In *Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2008.

[33] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving DRAM refresh-power through data partitioning. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[34] J. Meng, S. Chakradhar, and A. Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2009.

[35] Meta. `https://www.metavision.com/`, 2016. [Online; accessed 10-April-2016].

[36] Microsoft. Cortana. `https://www.microsoft.com/en-us/mobile/experiences/cortana/`, 2016. [Online; accessed 10-April-2016].

[37] R. G. Miller. *Simultaneous Statistical Inference*. Springer, 1981.

[38] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *International Conference on Software Engineering (ICSE)*, 2010.

[39] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *Static Analysis Symposium (SAS)*. 2011.

[40] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *Transactions on Embedded Computing Systems (TECS)*, 2013.

[41] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. Snnap: Approximate computing on programmable socs via neural acceleration. In *High Performance Computer Architecture (HPCA)*, 2015.

[42] C. Poynton. *Digital video and HD: Algorithms and Interfaces*. Elsevier, 2012.

[43] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener. Programming with relaxed synchronization. In *Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012.

[44] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *International Conference on Supercomputing (ICS)*, 2006.

[45] M. Rinard. Probabilistic accuracy bounds for perforated programs. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2011.

[46] M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2010.

[47] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman. Monitoring and debugging the quality of results in approximate programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[48] S. Russell and P. Norvig. Artificial intelligence: A modern approach. *Prentice Hall Press*, 1995.

[49] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. SAGE: Self-tuning approximation for graphics engines. In *International Symposium on Microarchitecture (MICRO)*, 2013.

[50] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[51] 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 *Programming Language Design and Implementation (PLDI)*, 2011.

[52] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. In *International Symposium on Microarchitecture (MICRO)*, 2013.

[53] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In *Programming Language Design and Implementation (PLDI)*, 2014.

[54] J. Sartori and R. Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. In *IEEE Transactions on Multimedia*. 2013.

[55] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: automatic software self-healing using rescue points. *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[56] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Foundations of Software Engineering (FSE)*, 2011.

[57] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *Dependable Systems and Networks (DSN)*, 2010.

[58] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: A language and runtime system for perpetual systems. In *Conference on Embedded Networked Sensor Systems (Sensys)*, 2007.

[59] D. Ungar, D. Kimelman, and S. Adams. Inconsistency robustness for scalability in interactive concurrent-update inmemory molap cubes. *Inconsistency Robustness (IR)*, 2011.

[60] J. S. Witte, R. C. Elston, and L. R. Cardon. On the relative sample size required for multiple comparisons. *Statistics in medicine*, 2000.

[61] T. Y. Yeh, P. Faloutsos, M. Ercegovac, S. J. Patel, and G. Reinman. The art of deception: Adaptive precision reduction for area efficient physics acceleration. In *International Symposium on Microarchitecture (MICRO)*, 2007.

[62] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *Principles of Programming Languages (POPL)*, 2012.