ASAC: Automatic Sensitivity Analysis for Approximate Computing

Pooja Roy[1], Rajarshi Ray[2], Chundong Wang[3], Weng-Fai Wong[1]

School of Computing, National University of Singapore[1], Singapore
National Institute of Technology, Meghalaya[2], India
Data Storage Institute[3], A*STAR, Singapore
{poojaroy@comp.nus.edu.sg, raj.ray84@gmail.com, wangc@dsi.a-star.edu.sg, wongwf@comp.nus.edu.sg}

Abstract

The approximation based programming paradigm is especially attractive for developing error-resilient applications, targeting low power embedded devices. It allows for program data to be computed and stored approximately for better energy efficiency. The duration of battery in the smartphones, tablets, etc. is generally more of a concern to users than an application's accuracy or fidelity beyond certain acceptable quality of service. Therefore, relaxing accuracy to improve energy efficiency is an attractive trade-off when permissible by the application's domain. Recent works suggest source code annotations and type qualifiers to facilitate safe approximate computation and data manipulation. It requires rewriting of programs or the availability of source codes for annotations. This may not be feasible as real-world applications tend to be large, with source code that is not readily available.

In this paper, we propose a novel *sensitivity analysis* that *automatically* generates annotations for programs for the purpose of approximate computing. Our framework, *ASAC*, extracts information about the sensitivity of the output with respect to program data. We show that the program output is sensitive to only a subset of program data that we deem critical, and hence must be precise. The rest of the data can be computed and stored approximately. We evaluated our analysis on a range of applications, and achieved a 86% accuracy compared to manual annotations by programmers. We validated our analysis by showing that the applications are within the acceptable QoS threshold if we approximate the non-critical data.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: - Automatic Programming; D.2.5 [*Software Engineering*]: Testing and Debugging - Code inspection and walk-throughs, Error handling and recovery

Keywords approximate computing, power-aware computing, automatic programming, sensitivity analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
LCTES '14, June 12–13, 2014, Edinburgh, United Kingdom.
Copyright © 2014 ACM 978-1-4503-2877-7/14/06…$15.00.
http://dx.doi.org/10.1145/2597809.2597812

1. Introduction

Approximate computing is a new programming paradigm that allows programs to trade-off accuracy of internal program data in favour of lower energy consumption. It is especially appealing to low-power embedded devices where energy efficiency is of serious concern. Further, there are many applications targeted to smartphones, tablets, etc. that are capable of tolerating inaccuracy while maintaining the desired quality of service (QoS). Many recent works have shown this to be a promising trade-off for current and future embedded platforms [2, 9, 13, 17, 25, 29].

In general, many programs contain specific parts that contribute to the correctness of the output and others that do not. A correct output usually lies within a Quality of Service range of the application. The parts of a program that do not affect the output beyond a tolerable extent are deemed *approximable*, while parts that are important are *non-approximable*. Depending on the application, the ratio between these two parts can vary significantly. The presence of approximable data in a program is the mainstay of the approximate computing paradigm. The non-approximable program data can be computed and stored in a high power mode, while the approximable regions in a low power mode [9]. Applications allowing such behaviour are called *error-tolerant*. Error-tolerance of applications running on devices prone to soft-errors is well studied [14, 16, 28]. However, in this new paradigm, instead of mitigating the errors, a controlled degradation of QoS due to the errors is allowed.

The challenge is how to discover the distinct approximable and non-approximable parts of a program *automatically*, so that adapting this approximation based programming paradigm is easier. Recent works have proposed source code annotations and type-qualifiers for programmers to indicate whether a variable (data) is error resilient, in other words, approximable [17, 25]. However, this implies rewriting or annotating source codes. This may be easily accomplished for small programs, but is difficult or infeasible for complex programs and legacy softwares. Other works have shown that program approximations can be achieved through algorithmic choices, runtime decision making frameworks, and on the architectural or device level [1, 2, 6, 27]. The provision of algorithmic choices too is the programmer's responsibility and the application is compiled using all the versions of a procedure. This is not only difficult when dealing with large applications having large numbers of procedures, it also inflates size of executables. Such consequences impede the usage of these solutions for embedded devices.

In this paper, we propose "ASAC" - *Automatic Sensitivity analysis for Approximate Computing*, a framework to *automatically* discover approximable data from a program. The main component of this framework is a specialized sensitivity analysis using statistical methods. Sensitivity analysis of parameters of mathematical mod-

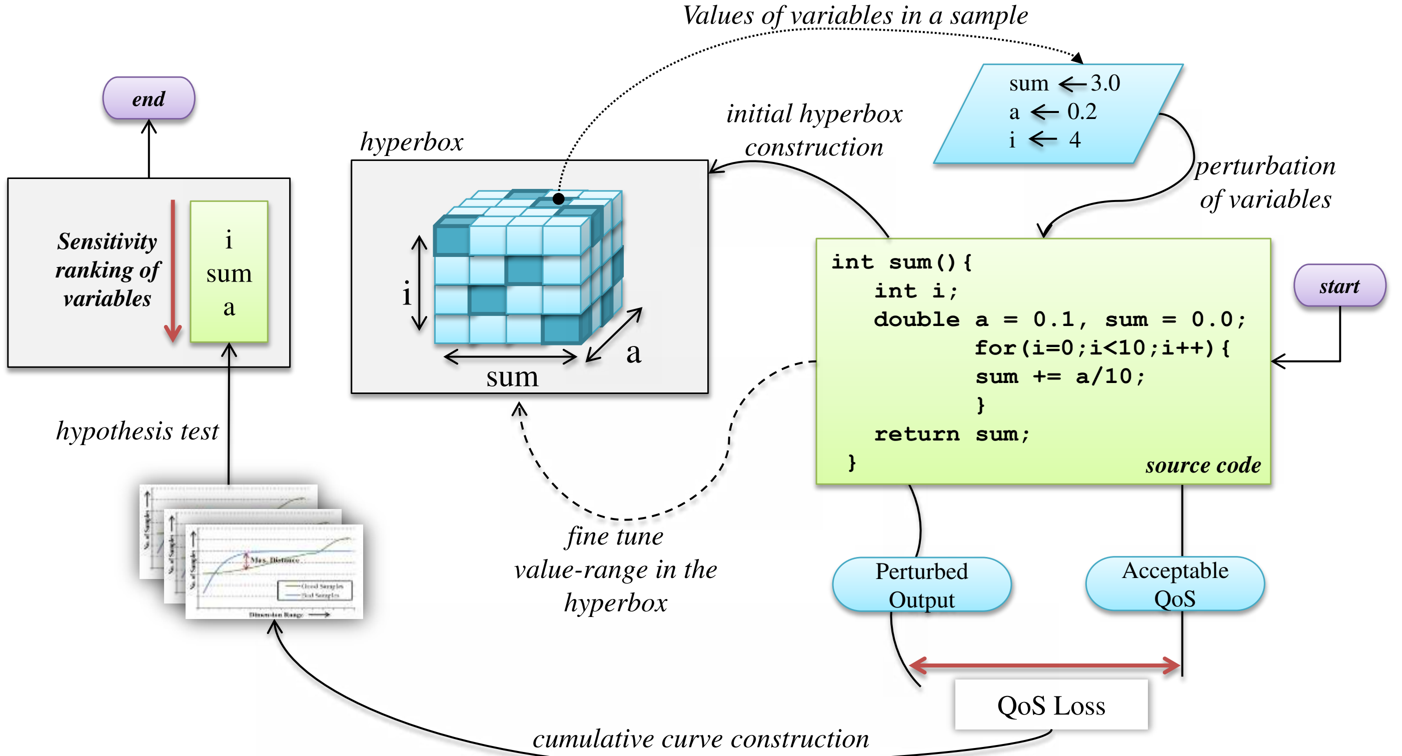


Figure 1: Overview of "ASAC" framework. Each box represents a step and the arrows are the dataflow between them. There is a information flow from Sampler back to the Hyperbox Construction to facilitate further optimization in range analysis.

els using statistical methods is known in literature [21]. Our contribution in this paper is the use of statistical methods for sensitivity analysis of program data. It consists of a random sampler and a hypothesis tester. The main idea of the analysis is to systematically perturb variables and then observe the resultant output sensitivity. Using the outputs from the probes, ASAC applies a hypothesis tester to check against a correct output, which by definition is one fulfilling an acceptable QoS threshold. The hypothesis test generates scores for each variable that ranks the variable's contribution to the output of the program. Based on the scores, the variables are classified as approximable or non- approximable.

ASAC is fully automatic and alleviates the programmer's involvement. With minor modification, it can also be applied to programs where the source code is not available. A direct application of this framework can be as a feedback system to a compiler, providing information about how the program's may be approximated. Moreover, ASAC can be used as a black-box tester to gain insight about the sensitivity of program output against program data. This would be valuable information for platforms susceptible to soft-errors, where instead of allowing the approximation, the sensitivity of the variables can be used as a metric to decide which data should be protected. We evaluated our analysis against a 'gold' standard where a programmer has made type-qualifier based annotations to programs to facilitate approximation [25]. We achieve 86% accuracy in determining approximable data with respect to this manually annotated baseline (MAB). In addition, to show the scalability and generality of our analysis, we apply it to bigger and more complex programs from MiBench and SPEC2006 benchmark suites. Our contributions in this paper are summarized as follows :

- The first automated software analysis that partitions program data for approximate computing based programming paradigm.
- A framework to discover program data that can be approximated without compromising the QoS of a given application.
- A black-box analysis that can test programs and order the variables in terms of their contribution to the correctness of the final output.

The rest of the paper is organized as follows: after a brief overview and motivation in Section 2, we will describe our framework in Sections 3 and 4. The evaluation of our analysis is presented in Section 5. Related works are explored in Section 6 and we conclude our paper in Section 7.

2. Overview

In this section we present a brief overview of our framework, ASAC. Our motivation for this work is twofold. The first is to alleviate the existing burden placed on the programmer in facilitating approximate computing. We aim to automatically analyze a program and identify data that can be approximated. In scenarios where annotating programs without programmer's knowledge is considered unsafe, ASAC can serve as the suggestive framework for annotating bigger and more complex programs. Programmers can then fine-tune ASAC's analysis results to obtain the final partitioning. In any case, it is obviously expensive, time consuming, and in some scenarios, infeasible to identify approximable and non-approximable data, and annotate the application completely manually. Moreover, for legacy softwares and other programs that has undergone significant changes over many versions, it may be diffi-

cult to understand the implications of approximated variables and their effect globally. Therefore, an automated analysis is indispensable for approximate computing in the large.

Our second motivation is to study the error-resilience of internal program data i.e. program variables, etc. Error-resilient program transformations has been well studied. However, all the existing works focus on approximating different components such as procedure approximation, input data approximation, control- flow based approximation etc. [4, 27]. Other works have studied the error-resilience of data in architectural components such as the arithmetic units, register files, etc. [6, 15]. Here, we are proposing a framework to analyse and approximate internal program data while maintaining an acceptable QoS according to the application.

The key idea is to systematically perturb the program variables and to observe its effect on program output. By quantifying the sensitivity of the output to the perturbations, we can discern program variables in terms of their contributions to the output. A variable that does not contribute to the correctness of the output or the functionality of the program beyond a certain extent is not considered as critical, and therefore can be approximated. Conversely, critical variables cannot be approximated and must be precise.

Figure 1 illustrates ASAC consisting of 3 main stages, namely *discovery*, *probe* and *testing*. In discovery stage, we extract the variables of a program along with the range of values that each can assume during the execution. The cartesian product of the variable range intervals defines an n-dimensional hyperbox. This hyperbox is the sample space for the statistical experiments performed by the sensitivity analysis module. Each dimension represents a variable and the corresponding edge of the hyperbox is the range of that variable. Therefore, the total number of dimensions in the hyperbox is determined by the number of variables in the program.

At the subsequent probe stage, we first divide the hyperbox into smaller hyperboxes of equal sizes. We select a subset of these smaller hyperboxes, the *samples*, and choose a number of points from among them. Each of these points are n-tuple coordinates containing the values of each variable at that point. These points are passed to the program and the values are forcefully assigned (perturbed) to corresponding variables during the execution by means of binary instrumentation. Due to the intrusion, the program output can be expected to be deviated from the correct output. Our aim is to measure this incorrectness. According to the difference between the QoS threshold of the application and the perturbed outputs, we mark each such sample as "good" (pass) or "bad"(fail).

Next, in the testing stage, a cumulative distribution curve is obtained by plotting the number of good or bad samples against the range of each dimension of hyperbox. The two curves undergo a hypothesis test that generates the maximum distance between them. A large distance between the curves means that the program output is very sensitive to the variable representing that dimension of the hyperbox. Conversely, a smaller distance implies the opposite. Each of the stages are described in details in the following sections.

3. Automated Analysis

In this section, we will describe the three stages in detail. First, we explain two main concepts integral to the discovery stage - range analysis and hyperbox construction. Range analysis is well studied. It is commonly used to detect integer overflows, etc. However, here we apply range analysis to estimate the values that a variable can assume during program's execution.

Definition *For each variable V_i in program under analysis, let value(V_i) be the value that V_i can assume during program execution. Then, range(V_i) = $[R_{i1}, R_{i2}]$, where $R_{i1} \leq value(V_i) \leq R_{i2}$. If $R_{i1} = \pm\infty$ or $R_{i2} = \pm\infty$, $range(V_i)$ is given by the datatype of V_i.*

Algorithm 1 Range Analysis

Input:
1: Program P, QoS Threshold Q
Output:
2: $R[n]$, where n $\leftarrow$ no. of variables in P

3: Initialize $rangeOf(V_i) = \emptyset \; \forall \; V_i$ in P
4: **for** each variable V_i in P **do**
5:     **if** $rangeof(V_i) = \emptyset$ **then**
6:         $var \leftarrow V_i$
7:         $R_i[2] \leftarrow$ RANGE_ANALYSIS(var) /* standard widening & narrowing operator based */
8:         **if** $R_i[0] \vee R_i[1] = \infty$ **then**
9:             **if** DATATYPE(var) = int_32 **then**
10:                 $R_i[0] \leftarrow -32767$ /* standard data */
11:                 $R_i[1] \leftarrow 32767$ /* range for int type */
12:             **else if** DATATYPE(var) = float **then**
13:                 $R_i[0] \leftarrow 0$ /* dummy range */
14:                 $R_i[1] \leftarrow 1$ /* that will shrink over runs */
15:             **else**
16:                 $/*$ handle all datatypes similarly $*/$
17:             **end if**
18:         **end if**
19:     **end if**
20: **end for**
21: **return** $R_i[]$

Variables	Datatype	Initial Range	Tuned Range
LineSadBlk0	double	[1 , 1]	[0.0 , 780.0]
P_A	int	[2048 , 2048]	[128 , 128]
P_E	int	[-32768 , 32767]	[34 , 244]
D_dis1	double	[1 , 1]	[-15.0 , 177.0]

Table 1: Ranges of some variables in H.264

This value range is essential for the construction of the hyperbox. We employ widening and narrowing operators based dataflow analysis to calculate the value ranges of the variables [23]. Algorithm 1 gives a pseudo-code description of our range analysis. In cases where the analysis is unable to generate a finite value range, we fine-tune the range based on the data type of the variable (line 9-12). For floating-point variables, we assume a dummy starting range of zero (line 13-14).

In order to extract the real value range, we have a information loop back (see Figure 1) from the sampler to hyperbox construction which makes it easy to get narrow and precise value-ranges from the profile runs of the program. Therefore, even if the dataflow analysis generates an infinite range for a variable, it is soon mitigated. This is shown in Table 1 with the examples of some of the variables in H.264. After calculating the ranges of the variables, we can construct the hyperbox.

Definition *An n-dimensional hyperbox $\mathbb{H}$ is the cartesian product of the range intervals of each of the n variables.*
$\mathbb{H} = [R_{11}, R_{12}] \times [R_{21}, R_{22}] \times \ldots \times [R_{n1}, R_{n2}]$*, where* $[R_{i1}, R_{i2}]$ *is the range of variable* (V_i)*.*

Figure 2 shows a conceptual diagram of hyperboxes. Each dimension represents a variable and thus with n variables it will have $n - dimensions$. The starting and ending point of each dimension is R_1 and R_2 of each variable i.e. the range. As the value range of a variable can narrow or widen over runs, the hyperbox may also shrink and grow. The shaded areas are called *samples*.

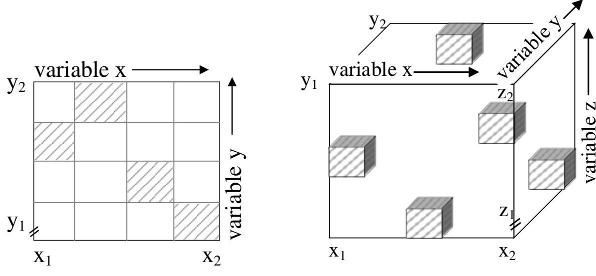


Figure 2: Example of 2 dimensional and 3 dimensional hyperboxes. A 2-variable program would generate a 2-D hyperbox and similarly, a 3-variable program would generate a 3-D hyperbox as shown.

These are small hyperboxes obtained by discretizing the edges, and selecting only a subset from among them. Discretization provides a finite sampling space from the original hyperbox which has infinitely many sample points. The finite sample space can then be sampled using any statistical sampler. In our framework, we have used the *Latin Hyperbox Sampling* (LHS) algorithm [18]. LHS ensures that the sampling is bias free and with a fairly well coverage of the sample space.

As shown in Figure 2, the 2-D hyperbox is discretized into equal sized grids, and only one sample from each row and column are qualified to be in the subset. For n-dimension, LHS selects only a subset of the samples based on their positioning. The complexity of this method depends on two factors - n, number of variables, i.e. the dimension of the hyperbox, and the constant k i.e. the discretization parameter. Empirically, the number of samples to be selected from a hyperbox can be defined as follows -

$$\text{Number of samples} = (\prod_{n=0}^{(k-1)} (k-n))^{n-1}$$

Next, in the first step of the probe stage, we choose m uniformly random points from each sampled hyperboxes. We will present a study of the effects of the constants in a later section. Each of these points are an n-tuple coordinate, where n is the number of variables in the program. For example, a point m_i from a sample s_i has the coordinates $(m_{i_1}, m_{i_2}, ...m_{i_n})$, where m_{i_1} is the value of variable V_1 at the point m_i. The points can be represented as a vector of real numbers. We use these vectors to introduce perturbation in the program execution by passing the values dynamically with an instrumentation tool. We call a program execution with the perturbed values a *probe run*. As the hyperbox was originally constructed by the value ranges of the variables, the perturbation for each variable lies within the range of values the variable is expected to assume during executions.

Definition *Let P_i be a vector of the outputs of all the probe runs of sample S_i, f_{obj} be an objective function, and θ is a constant threshold. If $f_{obj}(P_i) \geq \theta$ then designate P_i to be a "good" sample, else mark it as a "bad" sample. We define the objective function as*

$$f_{obj} = (\sum_{j=0}^{j=k} \omega(P_i))/k$$

where $\omega(P_i)$ = 1 if $P_i \geq T_{qos}$, otherwise $\omega(P_i)$ = 0. T_{qos} is the QoS threshold for the application given by the user.

Algorithm 2 illustrates the detailed steps involved in the construction of hyperbox and how points from among the samples are chosen. First, a hyperbox is built using the preliminary value ranges obtained from the range analysis (line 3-10). For each variable represented by a particular dimension (edge) of the hyperbox, the edge is discretized into k intervals (line 13-17). Therefore, we have $dim * k$ number of smaller hyperboxes after this step, where dim is the total number of variables and k is the discretization constant. A subset of these smaller hyperboxes are chosen using LHS to have a fair coverage of the ranges (line 18-22). The samples represent the set of values to be passed to the program in the probe runs. The perturbed outputs from all the probe runs are partitioned into two classes - "good" or "bad", based on the QoS threshold of the application.

Algorithm 2 Hyperbox Construction & Sampling

Input:
1: $Range[n][2]$, where n is no. of variables in program
2: k, discretization factor
Output:
3: $Vector[n][k]$, containing the values to be passed to program for perturbed run
4: **procedure** HYPERCUBE(Range[n][2],n)
5: Initialize $H \leftarrow \emptyset$
6: Initialize $dim = n$
7: **for** $i = 0$ to dim **do**
8: $H[i].leftdiagonal \leftarrow Range[i][0]$
9: $H[i].rightdiagonal \leftarrow Range[i][1]$
10: **end for**
11: **end procedure**

12: **procedure** LATIN HYPERCUBE SAMPLING(H[n],dim,k)
13: **for** $i = 0$ to k **do**
14: **for** $j = 0$ to dim **do**
15: $L = H[j].leftdiagonal$
16: $U = H[j].rightdiagonal$
17: $Interval_Size = (U - L)/k$
18: $Interval_Val =$ CHOOSERANDOM(i,j)
19: $LowLim = Interval_Val * Interval_Size$
20: $T[0][0] \leftarrow L + LowLim$
21: $T[0][1] \leftarrow L + LowLim + Interval_Size$
22: $Sample[i][j] \leftarrow Sample[i][j] \cap$ HYPERCUBE(T[1][2],j)
23: **end for**
24: **end for**
25: **return** $Sample^T$
26: **end procedure**

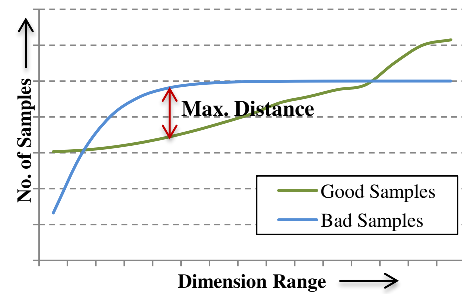


Figure 3: This graph is an example of how the CDFs are plotted and the hypothesis test's computes the distance metric using them. The two curves are plotted from the cumulative number of samples that are marked "good" and "bad" based on the QoS.

From all the samples marked as either good or bad (0 or 1), we construct a cumulative curve for each dimension of the hyperbox. The number of good samples is counted, and plotted against the range of that dimension. Similarly, a second curve is obtained by counting the samples marked as bad. These two curves are regarded as two cumulative distributions obtained from the perturbed program runs.

Definition *Let Sen_i denote the sensitivity score for a variable V_i. Let $f_{good}V_i$ and $f_{bad}V_i$ be the two cumulative distribution function (CDF) for variable V_i. Then, $Sen_i = max_x|f_{good}V_i(x) - f_{bad}V_i(x)|$, where x is a point in the value range of the variable V_i at which the CDFs are calculated.*

Intuitively, the distance between the two curves denotes the contribution of this variables towards the program output. We apply the *Kolmogorov-Smirnov* hypothesis test [21] to calculate the maximum distance between the two curves. This is called the d-statistics, and it translates to the sensitivity ranking: the higher the distance, the higher is the sensitivity of the output to this variable, and vice-versa. Figure 3 shows an example of the cumulative curves, and the maximum distance between them. A detailed step-by-step description of the generation of the sensitivity scores is given in Algorithm 3. First, the program probe step is detailed in lines 3-15. The procedure receives the vector of values from the hyperbox as input and runs the program by forcefully assigning these values to the variables. Each program run produces a result that is stored to be compared for QoS at a later stage. In this procedure, the hypercube is also updated with fine-tuned range of the variables. Next, in the hypothesis test procedure, an error is calculated from the obtained result and the original result of the program (line 18). This error is used to mark a sample as good or bad. Following this marking, considering all the samples from the hyperbox, a cumulative graph is plotted against each dimension (lines 27-33). Two curves are obtained for each dimension of the hyperbox and they are passed to the KS-Test for the distance metric (line 35).

4. Optimizations

4.1 Discretization Constant

Our proposed analysis has one tunable parameter, the discretization constant, k. This determines the size of the samples for each dimension in the hyperbox. In other words, all the value ranges of the variables are divided up using this constant k so as to reduce the value space (see Algorithm 2). The completion time of the analysis is affected by this parameter. A larger value will cause the analysis to take a longer time to complete because the hyperbox is divided into smaller grids. However, the sensitivity scores obtained from the analysis is not affected by the value of k as shown in Table 2. Thus, we can conclude that the sensitivity of program output with respect to its variables is a characteristic of the program. For our evaluation, we tested with $k = 10, 50, 100, 200$.

Figure 4 shows the total time taken by ASAC to complete its analysis. As shown, when $k \leq 100$, ASAC takes longer time to rank the variables. Table 2 shows the percentage of total variables marked as approximate with two different k values. The percentages for $k = 5$ and $k = 200$ are same as that for $k = 10$ and $k = 100$, respectively. The difference between $k = 10$ and $k = 100$ is attributed to the fine tuning of the ranges of the variables. As the percentages are averaged over 20 runs, different program paths will result in different fine-tuning of the variable ranges. Nonetheless, the difference of percentage of variables marked as approximate shows no significant variation over the values of k as shown in Table 2.

There is another constant *m*, which determines how many points will be chosen from within one sample to perturb the program. We

Algorithm 3 Sensitivity Ranking

Input:
1: $Vector[n][k]$, containing the values to be passed to program for perturbed run
2: Q, QoS Threshold
Output:
3: $SenScores[n]$, sensitivity scores for variables
4: **procedure** PROGRAM PROBE(Vector[n][k])
5: Initialize $Values[n] \leftarrow \emptyset$
6: Initialize $dim = n$
7: **for** $j = 0$ to k **do**
8: **for** $i = 0$ to dim **do**
9: $Values[i] \leftarrow Vector[i][j]$
10: **end for**
11: $Output[j] \leftarrow$ program executed with $Values[]$
12: Update hypercube
13: **end for**
14: **return** Output[]
15: **end procedure**

16: **procedure** HYPOTHESIS TEST(Output[],Q)
17: **for** $i = 0$ to k **do**
18: $err =$ GETERRORFUNCTION(Output[i])
19: **if** $err \leq Q$ **then**
20: $Good[i][] = Values[]$
21: **else**
22: $Bad[i][] = Values[]$
23: **end if**
24: **end for**
25: **for** $i = 0$ to dim **do**
26: $j \leftarrow R_i[0]$
27: **while** $j \neq R_i[1]$ **do**
28: **if** $j \in Good[i][]$ **then**
29: $C_{good}[j] + +$
30: **else if** $j \in Bad[i][]$ **then**
31: $C_{bad}[j] + +$
32: **end if**
33: $j+ = Interval_Size$
34: **end while**
35: $SenScores[i] \leftarrow$ KS_TEST(C_{good}, C_{bad})
36: **end for**
37: **end procedure**

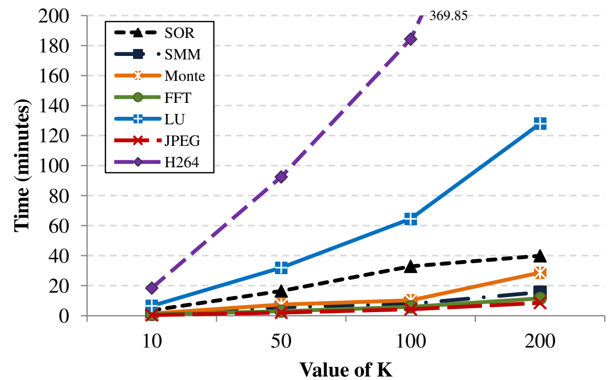


Figure 4: Total time (in minutes) taken by ASAC to produce the variable rankings with varied values of the parameters k while $m = 2$.

observe an interesting trend in the relationship between k and m. As the samples are small in size with a high value of k, increasing the value of m, i.e. choosing many points within a narrow range, results in passing similar values for probing. Therefore, the value of this constant m has no significant impact on the variable ranking when k is high. Nonetheless, a high value of both k and m will translate to higher running time for our analysis.

When k is small, the value of m has an impact on the variables' ranking. A small value of k and m will result in sampling a few representatives from a large hyperbox causing poor coverage of the sample space. This behaviour is accentuated in bigger programs, such as JPEG and H.264. However, it is important to have perturbations with values of variables that are uniformly distributed over its range. Therefore, for our experiments we used $m = 5$ and $k = 100$.

4.2 Perturbation Points

In the probe stage of our framework, we force variables to assume values chosen from the hyperbox. We use the dynamic instrumentation tool PIN [20] to inject the values at runtime. There are two important issues that we would like to discuss here.

First, it is a challenge to identify program points where the variables are perturbed. For example, if a perturbation is introduced at a point where a variable is first *used* after being *defined*, then the effect on output will be different than if the perturbation is introduced at a later point. In the former, the error might propagate and accumulate, resulting in a large deviation from correct output. On the other hand, the error might get masked by further arithmetic operations on the variables [27]. In our implementation, we introduce the perturbations at the first usage of a variable after it is defined. Nonetheless, it would be interesting to study the effects of the perturbation at other program points.

The second challenge is in injecting error into loop structures. It is difficult to force values into loop variables because of its iterative nature. Our aim is to perturb a variable to see the effect on the output. However, if the loop factor is high, then injecting the perturbation at every iteration becomes too aggressive. Instead we chose to perturb only a subset (25%) of the loop iterations. This technique is analogous to the concept of *loop-perforation* [29].

Benchmarks	Total Decls	Data Approximable(%)			
		$k = 2$		$m = 5$	
		$m = 2$	$m = 10$	$k = 10$	$k = 100$
SOR	28	28	28	28	28
SMM	29	27	27	27	27
Monte	15	33	33	33	33
FFT	85	32	35	36	35
LU	150	6	9	9	10
JPEG	1174	6	10	11	11
H.264	11857	7	15	16	16

Table 2: Percentage of variables marked as approximable by ASAC with different values of k and m.

4.3 Instrumentation & Testing

ASAC involves ranking the variables using their identifiers, i.e. names, which are not easily accessible after the code generation, especially at runtime. Therefore, it is difficult to pass the perturbation values to the program during the probe runs.

To force a sample value into a program, we implemented a compile-time pass that will inject additional code at the appropriate program point in the code to read the value to be forced into a variable from a file, and perform the write of that value into the variable. We also found that for larger applications, it was easier to use the PIN tool to inject such values - provided they are not bound to registers - into variables using their virtual addresses. In the actual implementation, we used a combination of both.

In the testing and evaluation of ASAC, we adapted the *bitflip error model* used by many prior works [3, 4, 15] to introduce errors into the application. A bitflip error essentially means that one or more bits within a data toggles one or more times during execution of the program, inducing an error. We used the same two techniques described above except that in the testing and evaluation, instead of forcing a targeted variable to take a certain value, we choose a (uniformly) random bit among the 16 lower bits of its current, and toggle it. There are many other error models available in literature, we chose bitflip because it is fairly simple to understand and model. Nonetheless, more complicated error models could also be used.

5. Evaluation

We evaluated ASAC against a manually annotated baseline (MAB) that uses type-qualifiers [25]. The authors of that paper kindly provided us with benchmarks from SciMark2 [22] that had such annotations made. We also apply ASAC to two benchmarks from SPEC2006 [30] and MiBench [11] to test its scalability. To measure the QoS loss due to approximation, we defined the error metric for each application, shown in Table 3. For FFT, LU and SOR, we use the mean squared error between the correct answer and the approximated output to quantify the degradation. For applications like SparseMatMult and MonteCarlo, we measure the normalized difference i.e. 0 if the approximated output is equal to correct output and 1 if not. For JPEG and H.264, we use the signal-to-noise ratio (SNR). The error estimation module as well as the QoS threshold is deemed to be provided by the user for our analysis. This makes it easy and portable.

Application	Benchmark	Error Metric	LOC
SOR	SciMark2	Mean Square Error	36
SparseMatMult	SciMark2	Normalized difference	38
MonteCarlo	SciMark2	Normalized difference	59
FFT	SciMark2	Mean Square Error	168
LU	SciMark2	Mean Square Error	283
JPEG	MiBench	SNR	30781
H.264	SPEC2006	SNR	46190

Table 3: Description of all the benchmarks used for evaluation.

Comparison with Manually Annotated Baseline (MAB). Table 4 shows the detailed comparison of ASAC with MAB. We shall examine the *precision, recall and accuracy* metrics of these experiments.

Precision measures how frequent a variable marked by ASAC to be approximable is also annotated as approximable in the MAB. Empirically it is defined as -

$$\frac{tp}{tp+fp}$$

where 'tp' and 'fp' are the 'true positive' and 'false positive' in Table 4, respectively. The former are those variables found to be 'approximable' in both ASAC and MAB. The latter are variables that ASAC declared to be 'approximable' but were annotated as 'non- approximable' in MAB. ASAC achieved a precision of 75%. The 25% loss in precision is due to the fact that our framework is more optimistic in marking variables as approximable.

Benchmarks	True Positive(tp)	False Positive(fp)	False Negative(fn)	True Negative(tn)	Precision	Recall	Accuracy
SOR	5	0	1	2	0.83	1.00	0.88
SMM	1	0	1	6	0.50	1.00	0.88
Monte	2	0	1	2	0.67	1.00	0.80
FFT	15	2	2	12	0.88	0.88	0.87
LU	7	1	1	5	0.88	0.88	0.86
				Average	0.75	0.95	0.86

Table 4: Comparison of *ASAC* with "EnerJ" [25].

Recall measures the robustness of our analysis. It is the complement of the percentage of variables our analysis mistakenly classifies a variable as non-approximable while MAB has annotated it as approximable, defined as follows -

$$\frac{tp}{tp+fn}$$

where 'fn' is 'false negative', variables that are marked as 'non-approximable' as ASAC but annotated as 'approximable' by MAB. These are the cases where our analysis fails to exploit approximable variables. Our analysis shows a high recall value of 95%. Finally, **accuracy** is a metric that combines precision and recall, and quantifies how much can we match the classification by MAB. It is defined as -

$$\frac{tp+tn}{tp+tn+fp+fn}$$

where 'tn' are the 'true negatives', i.e., the variables that both ASAC and MAB agree are non-approximable.

We achieve a high accuracy of 86%, using ASAC's fully automatic approach. The accuracy can be improved further by optimizations discussed in Section 4.

Error Measurement. In order to quantify the error due to approximation of program data, we evaluated different levels of error injection: two levels in JPEG, and three levels for H.264. First, in the `Mild` injection, errors are injected to only 50% of the variables marked as approximable. This half is chosen from the lower ranked variables (lower sensitivity scores) among those that are marked as approximable. Bitflip errors were injected into these variables during runtime.

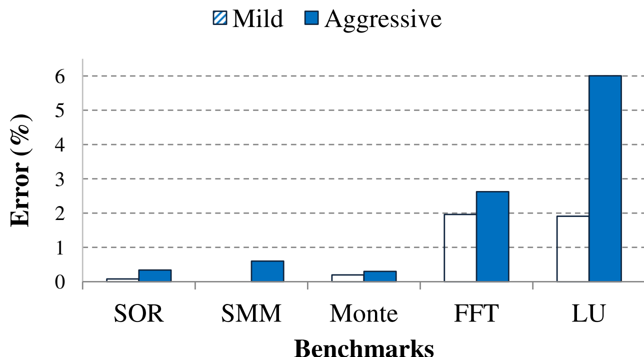


Figure 5: Percentage of error after approximating program data. The two bars are different error percentage after approximating either one-third or all the data that are classified as approximable by ASAC.

Second, in the `Aggressive` injection, errors are injected to all the variables identified as approximable. For H.264 that has a large number of variables, we created one more level of inject - `Medium`. For this benchmark, we chose the lowest scored one-third as the `Mild` injection, 60% for `Medium` and 100% (all) for `Aggressive`. Figure 5 shows the error percentages for the SciMark2 applications under `Mild` and `Aggressive` error injection. Figure 6 shows the result when `Mild` and `Aggressive` error injections were applied to the JPEG benchmark. We applied error injection to the encode and decode steps separately to show the effect of error accumulation. In the Figure 6(e) the errors are aggravated as it takes Figure 6(d) as its input which already contained the errors injected in encode step. Therefore, the `Aggressive` approximation for decode step is actually more severe than what it would have been if taken in isolation. Table 5 shows the approximation results for all the `Mild`, `Medium` and `Aggressive` applied to H.264.

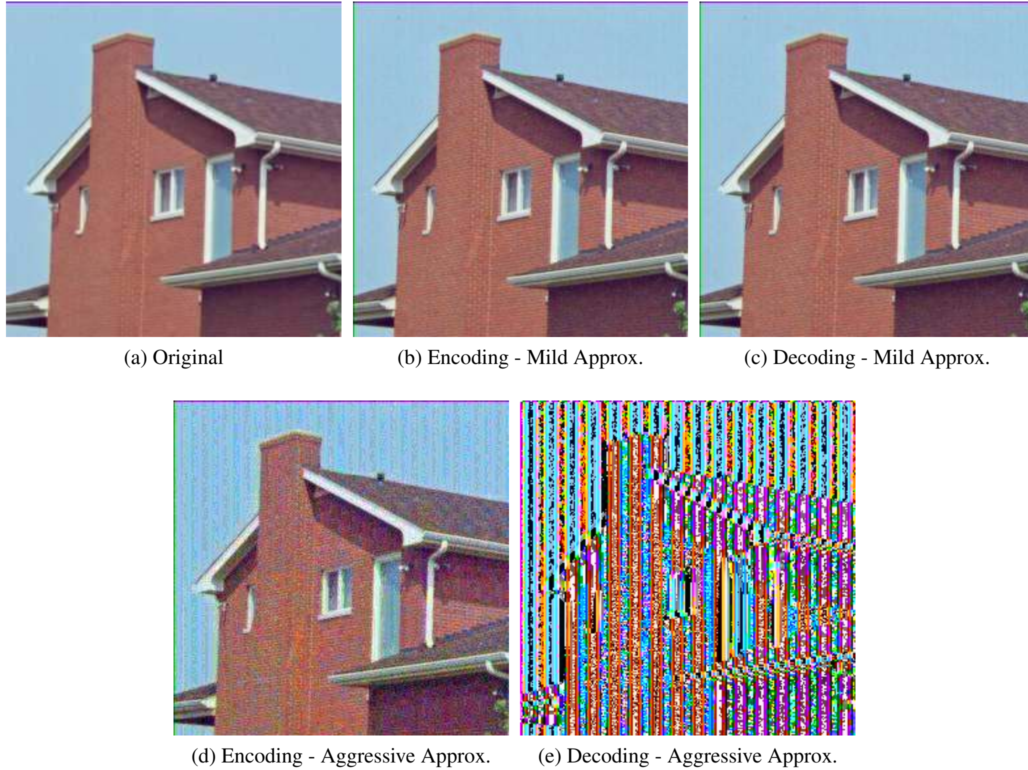
H.264	SNR_Y	SNR_U	SNR_V	BitRate
Correct	36.67	40.74	42.31	149.62
Mild	36.69	37.64	37.65	146.6
Medium	34.05	36.92	36.79	147.12
Aggressive	29.78	32.89	32.99	146.03

Table 5: H.264 Approximation Results

Further Studies on JPEG and H.264. As we do not have manual annotations for JPEG and H.264 benchmarks, we studied the effect of injecting errors into the variables that ASAC has marked as non- approximable. Essentially, there were two scenarios. First, when `Aggressive` error injection was applied to those variables deemed non-approximable (i.e., precise), the output of the JPEG benchmark was a corrupted image file, while the H.264 benchmark simply terminated pre-maturely with segmentation fault. This is because ASAC marks all pointers and memory addresses as non-approximable, hence an `Aggressive` error injection into memory addresses naturally resulted in crashes. Next, we tried to inject errors only into variables that ASAC has marked as non-approximable and are not memory addresses. Figure 7 shows the encode and decode outputs of JPEG. It clearly shows that ASAC is able to correctly mark not only approximable data, but also non-approximable data. For H.264, even a `Mild` error injection into non-pointer variables led to the application crashing.

6. Related Works

Emerging complex embedded devices generally face strict energy constraints. Users want to run a huge variety of application on their smartphones, tablets, etc. and expect a longer battery life. One way to achieve this is to trade-off the QoS.



(a) Original (b) Encoding - Mild Approx. (c) Decoding - Mild Approx.

(d) Encoding - Aggressive Approx. (e) Decoding - Aggressive Approx.

Figure 6: JPEG benchmark with various level of approximations separately in *Encode* and *Decode* stages. Image (a) is the original image. Images (b) and (c) are result of introducing mild approximation (in 30% of the variables). Images (d) and (e) are result of introducing aggressive approximation (in all the variables that are approximable).

There are many popular applications in commercial embedded devices that do not require a strict QoS as long as they meet an acceptable threshold [16]. Building on this idea, approximate computing has gained much attention. It allows programs to relax their accuracy in order to save on energy consumption. There are also many works focused on mitigating soft-errors in programs [14, 16, 28].

6.1 Approximation in Programs

Recently there has been proposals on how to allow a disciplined approximation to relax the accuracy of a program and reduce energy consumption as a consequence [2, 9, 17, 25, 26, 33]. Approximation is achievable at different levels of abstraction such as code approximation, program approximation, approximate computer architectures and device level approximations. Rinard proposed a probabilistic bound on erroneous output and thus making programs robust [24]. Furthermore, in [12] they suggested a trade-off in accuracy by a loop perforation technique to save energy by reducing computation.

Baek et.al. [2] also suggested a similar trade-off by proposing a loop and function approximation framework. In their work, the programmers are expected to provide multiple versions of a function or a loop structure. The framework consists of a calibration that generates a QoS model and allows a graceful QoS loss during runtime to save energy. However, this solution places a demand on the programmer's expertise and involvement. With the popularity of open-source application development for embedded devices, it is generally not feasible to request multiple versions of a code to allow approximate computing. In addition, compiling (or re-compiling, in case of legacy softwares) a program with extra versions of functions and loops would result in code bloat and larger executables which is not suitable for tight budget and low power devices.

In EnerJ, Sampson et.al. [25] proposes a type-qualifier based programming paradigm to facilitate approximation of program data. This ensures safety in terms of maintaining a distinction between approximate and precise computation of program data. Only with explicit programmer's endorsements, a conversion from precise to approximate or vice-versa is allowed. It provides an exclusive compiler to generate instructions for the underlying dynamic voltage scaling-based hardware called "Truffle" to switch between high and low power modes [9].

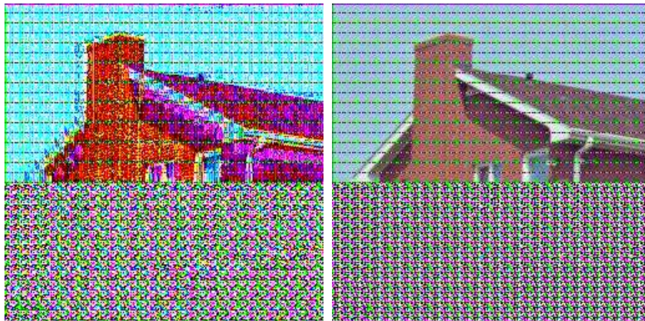
Carbin et.al. [4] proposed a technique that classifies code regions into approximable and critical by a training method that uses fuzzed input data. Depending on the program path taken by different inputs, it is able to identify critical program regions and approximable regions. Many works propose program transformations and code generation techniques to allow approximate computation. The motivation in these works is to save computation power i.e. loop iterations, floating point operations etc. One such method is known an "loop perforation" where loop iterations are skipped in order to save computation which results in approximation in the output [29]. Misailovic et.al. [19, 32] proposes probabilistic accuracy tests to allow for program level approximations.

However, in many works exploring accuracy energy trade-off [2][25], one common drawback is the programmer's involvement, and the issue of scalability. ASAC tries to address both.

6.2 Approximation in Hardware Devices

There are many other works investigating and designing architectural or device level approximation infrastructures [5, 7, 31].

Chippa et.al. [8] presented a work on characterizing error resilience in applications based on approximate adders. They also proposed "Impact", an approximate adder circuit that saves energy by approximating addition operations [10]. Liu et.al. [17] proposed a DRAM refresh mechanism that protects critical data and approximates non-critical data to save refresh energy. There has also been research works exploring program level error resilience that allows safe approximation. Shafique et.al. [27] proposes a technique to discover errors that are masked by program flow and operations on data. This indicates an inherent error resilience and approximation capability of a program. However, this is based on static code analysis and thus is not accurate as a whole program optimization framework. Error concerning only statically allocated data and compile-time inferable computation is exposed to this technique. Program data that are dynamically allocated or are influenced by runtime computations are hard to analyse. ASAC is a software testing based framework and thus, is able to analyse all kinds of program variables.



(a) Encoding - Mild Approx. (b) Decoding - Mild Approx.

Figure 7: JPEG benchmark with errors in data that are marked as "Precise" by ASAC.

7. Conclusion

In this paper, we present ASAC, a framework to automatically classify internal program data as approximable and non-approximable. We propose a novel sensitivity analysis that makes use of statistical sampling in performing a controlled perturbation based program testing. We are able to achieve 86% accuracy in identifying approximable data as compared to a manually annotated baseline. We also show that ASAC is scalable, and is able to analyze large applications such as JPEG and H.264. To the best of our knowledge, our work is the first to propose such an automated framework for approximate computing.

Our experimental results show that using our annotations to approximate program data resulted in program outputs that are within the acceptable QoS thresholds. ASAC is easy to adapt in either a compilation or a software testing framework. In addition, it can be used to provide suggestive annotations for large-scale programs that are difficult to annotate manually.

As a part of future work, ASAC can be extended to comprise more complex analysis and study sensitivity of program data across software versions. We expect ASAC to be a key contribution as the first automatic framework to classify program data in the field of approximate computing, which will grow as energy efficiency demands become more prevalent.

Acknowledgments

The research reported here was supported in part by the Singapore Ministry of Education Tier 2 Research Grant MOE2010-T2-1-075, and the A*STAR PSF Research Grant 102-101-0028. We also thank Manmohan Manoharan for valuable reviews.

References

[1] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 85–96, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL http://dl.acm.org/citation.cfm?id=2190025.2190056.

[2] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. . URL http://doi.acm.org/10.1145/1806596.1806620.

[3] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 453–462, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. . URL http://doi.acm.org/10.1145/2254064.2254118.

[4] M. Carbin and M. C. Rinard. Automatically identifying critical input regions and code in applications. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 37–48, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-823-0. . URL http://doi.acm.org/10.1145/1831708.1831713.

[5] V. Chippa, A. Raghunathan, K. Roy, and S. Chakradhar. Dynamic effort scaling: Managing the quality-efficiency tradeoff. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 603–608, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0636-2. . URL http://doi.acm.org/10.1145/2024724.2024863.

[6] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar. Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 555–560, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0002-5. . URL http://doi.acm.org/10.1145/1837274.1837411.

[7] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar. Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 555–560, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0002-5. . URL http://doi.acm.org/10.1145/1837274.1837411.

[8] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 113:1–113:9, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2071-9. . URL http://doi.acm.org/10.1145/2463209.2488873.

[9] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 301–312, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. . URL http://doi.acm.org/10.1145/2150976.2151008.

[10] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy. Impact: Imprecise adders for low-power approximate computing. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ISLPED '11, pages 409–414, Piscataway, NJ, USA, 2011. IEEE Press. ISBN 978-1-61284-660-6. URL http://dl.acm.org/citation.cfm?id=2016802.2016898.

[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. WWC '01, 2001.

[12] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. In *MIT Technical Report*. MIT, 2009. URL http://hdl.handle.net/1721.1/46709.

[13] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 199–212, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. . URL http://doi.acm.org/10.1145/1950365.1950390.

[14] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler-assisted soft error detection under performance and energy constraints in embedded systems. *ACM Trans. Embed. Comput. Syst.*, 8(4):27:1–27:30, July 2009. ISSN 1539-9087. . URL http://doi.acm.org/10.1145/1550987.1550990.

[15] J. Lee and A. Shrivastava. Static analysis to mitigate soft errors in register files. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1367–1372, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association. ISBN 978-3-9810801-5-5. URL http://dl.acm.org/citation.cfm?id=1874620.1874949.

[16] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Mitigating soft error failures for multimedia applications by selective data protection. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 411–420, New York, NY, USA, 2006. ACM. ISBN 1-59593-543-6. . URL http://doi.acm.org/10.1145/1176760.1176810.

[17] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 213–224, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. . URL http://doi.acm.org/10.1145/1950365.1950391.

[18] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 42(1):55–61, Feb. 2000. ISSN 0040-1706. . URL http://dx.doi.org/10.2307/1271432.

[19] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Trans. Embed. Comput. Syst.*, 12(2s):88:1–88:26, May 2013. ISSN 1539-9087. . URL http://doi.acm.org/10.1145/2465787.2465790.

[20] T. Naughton, W. Bland, G. Vallee, C. Engelmann, and S. L. Scott. Fault injection framework for system resilience evaluation: Fake faults for finding future failures. In *Proceedings of the 2009 Workshop on Resiliency in High Performance*, Resilience '09, pages 23–28, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-593-2. . URL http://doi.acm.org/10.1145/1552526.1552530.

[21] S. Palaniappan, B. Gyori, B. Liu, D. Hsu, and P. Thiagarajan. Statistical model checking based calibration and analysis of bio-pathway models. In A. Gupta and T. Henzinger, editors, *Computational Methods in Systems Biology*, volume 8130 of *Lecture Notes in Computer Science*, pages 120–134. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40707-9. . URL http://dx.doi.org/10.1007/978-3-642-40708-6-10.

[22] R. Pozo and B. Miller. Scimark 2.0. www.math.nist.gov/scimark2/.

[23] F. M. Quintao Pereira, R. E. Rodrigues, and V. H. Sperle Campos. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-1-4673-5524-7. . URL http://dx.doi.org/10.1109/CGO.2013.6494996.

[24] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 324–334, New York, NY, USA, 2006. ACM. ISBN 1-59593-282-8. . URL http://doi.acm.org/10.1145/1183401.1183447.

[25] 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 *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. . URL http://doi.acm.org/10.1145/1993498.1993518.

[26] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 25–36, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2638-4. . URL http://doi.acm.org/10.1145/2540708.2540712.

[27] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel. Exploiting program-level masking and error propagation for constrained reliability optimization. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 17:1–17:9, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2071-9. . URL http://doi.acm.org/10.1145/2463209.2488755.

[28] A. Shrivastava, J. Lee, and R. Jeyapaul. Cache vulnerability equations for protecting data in embedded processor caches from soft errors. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '10, pages 143–152, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-953-4. . URL http://doi.acm.org/10.1145/1755888.1755910.

[29] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0443-6. . URL http://doi.acm.org/10.1145/2025113.2025133.

[30] SPEC-CPU2006. Spec benchamrks. www.spec.org/cpu2006/.

[31] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 1–12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2638-4. . URL http://doi.acm.org/10.1145/2540708.2540710.

[32] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 441–454, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. . URL http://doi.acm.org/10.1145/2103656.2103710.

[33] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 441–454, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. . URL http://doi.acm.org/10.1145/2103656.2103710.