Impact of Software Approximations on the Resiliency of a Video Summarization System

Radha Venkatagiri† Karthik Swaminathan‡ Chung-Ching Lin‡ Liang Wang¶
Alper Buyuktosunoglu‡ Pradip Bose‡ Sarita Adve†
†University of Illinois at Urbana-Champaign ‡IBM Research ¶University of Virginia
†{venktgr2, sadve}@illinois.edu ‡{kvswamin, cclin, alperb, pbose}@us.ibm.com ¶lw2aw@virginia.edu

***Abstract*— In this work, we examine the resiliency of a state-of-the-art end-to-end video summarization (*VS*) application that serves as a representative emerging workload in the domain of real time edge computing. The *VS* application constitutes key video and image analytic elements that are processed by embedded systems aboard unmanned aerial vehicles (UAVs).**

Real-time performance and energy constraints motivate the consideration of approximations to the *VS* algorithm. However, mission-critical UAV applications also demand stringent levels of resilience to soft errors that are exacerbated with higher altitude. In this work, we study the effects of three different types of software approximations on the application level resiliency (to soft errors) of the *VS* algorithm. We show that our approximations yield significant energy savings (up to 68%), with commensurate improvement in performance, without a degradation in the application resilience. Further, by proposing a novel quality metric (appropriate for the UAV vision analytics domain) for the summarized video output, we show that even though the rate of Silent Data Corruptions (SDCs) increases slightly (<2%), the impact of these SDCs on output quality is limited. Thus, we conclude that software approximation can be utilized to achieve significant gains in performance and energy without affecting application resiliency.

I. Introduction

Real time edge computing [1], [2] is a rapidly growing field where real time data processing and other compute services are pushed away from centralized points to the logical extremes or *edge* of a network. This reduces the communication bandwidth needed between edge devices (e.g., sensors) and the central data center by performing analytics and knowledge generation at or near the source of the data. One of the key enablers of this trend is the presence of simultaneously high-performance and energy-efficient embedded systems that can be used to do computing in devices that are at the edge of the network. Real time edge computing has many applications which are both military and civilian in nature, such as Unmanned Aerial Vehicles (UAV), connected cars, industrial robotics, etc.

Figure 1 illustrates a real-life application of edge computing. Here, a swarm of UAVs, supported by a terrestrial server at the back-end, carry out tasks such as surveillance of hostile targets or rescue and recovery in the event of natural disasters. The UAVs communicate data and other analytics with the ground servers through wireless connections whose bandwidth, security and reliability might vary depending on physical and environmental factors. Hence, for real-time critical tasks, it is increasingly becoming essential that each UAV be equipped as a highly efficient mobile embedded system that can locally perform essential real-time computing tasks. One such task that is often performed locally aboard the UAV is *Video Summarization* [3]. This task involves extracting concrete context from the input video stream – captured by the many cameras on the moving UAV surveying a wide area – and summarizing it, usually in the form of a panoramic image. The panoramic image can then be transfered to a central ground server for further processing, for say, tracking or identifying rescue targets.

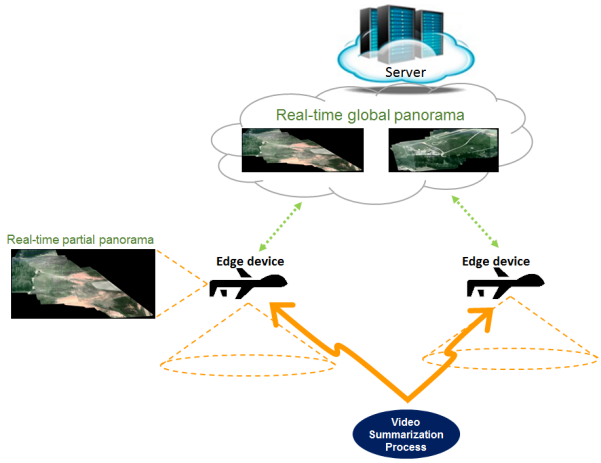


Fig. 1: Co-operative swarm of UAVs engaging in computation for real-time applications.

Edge computing platforms, such as that described above, are often deployed in rugged terrains with harsh environmental conditions and must satisfy the following requirements: a) ensure *high performance* to meet real time deadlines, particularly for mission-critical applications, b) be *energy efficient* to enable long range computing, and c) be *resilient* while operating in harsh environments subject to sharp variations in temperature, altitude and weather conditions, and tolerate glitches in input and output [4].

Approximate Computing [5], [6] is increasingly gaining traction as a viable approach for high performance and energy

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA), by the National Science Foundation under Grant CCF-1320941, by the Center for Future Architectures Research (C-FAR) and the Applications Driving Architectures (ADA) center, one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. This paper is: Approved for Public Release, Distribution Unlimited.

efficiency. Approximate computing environments allow deliberate, but controlled, relaxation of correctness and trade-off computational accuracy for improvements in performance and energy. Many edge computing applications involve processing sensory signals (image, audio, etc.) which can inherently tolerate inaccuracies in data and/or computation without compromising overall mission targets and goals. This presents an opportunity to redesign these algorithms to incorporate approximate computing with the goal of meeting stringent performance and energy targets (requirements (a) and (b) from above) under specified constraints.

However, while most approximate computing techniques have in-built metrics and techniques to guarantee a certain output quality, it is not clear how they work in the face of sources of vulnerability in the processor, such as soft errors, voltage noise and aging phenomena. Further, these effects can be exacerbated (increased probability of radiation strikes at high altitudes in UAVs) when the approximate computing paradigm is adapted to the harsh conditions that these systems encounter during their operation. For successful deployment in edge computing environments, it is critical to ensure that the application of approximate computing techniques which yield performance and energy improvements not degrade the overall system resiliency (requirement (c) from above).

This work focuses on studying the interaction between software approximation and the application's resiliency to soft errors (henceforth referred to as *application resiliency* or simply *resiliency*) and to our knowledge is the first work to do so. To demonstrate this interaction we analyze a state-of-the-art end-to-end video summarization application [3] which represents a typical and key vision analytics workflow executed by embedded systems on-board UAVs.

In particular, we make the following contributions:

- We study the application resiliency of an end-to-end video summarization application (henceforth termed as *VS* for brevity) that serves as a representative workload for on-board UAV processing. Specifically, we study the application's resilience to radiation-induced soft (transient) errors, by performing runtime architectural fault injection experiments. We perform all our analyses across two distinct inputs that realistically portray the different types of input video stream captured by cameras on the UAV.
- Performing resiliency analysis on a full, long-running end-to-end workflow is more expensive (in time and compute) than analyzing individual smaller kernels that together constitute the larger work-flow. We examine this trade-off by estimating the resiliency of individual representative kernels or *hot* functions in the *VS* application. We show that the hot functions are sub-optimal at capturing the behavior of the full application, thus motivating the need to develop and evaluate realistic applications with a full end-to-end workflow.
- We characterize the performance and energy of three different software approximation techniques applied to the VS algorithm. We show that the approximations yield significant speedup and energy savings (up to 68%) without compromising the quality of the panoramic image output.
- We examine the resiliency of approximate *VS* algorithms. We find that the approximations yield similar resiliency profiles to the baseline (precise) algorithm and in the worst case lead to a slight increase in Silent Data Corruption (SDC) rates (up to 2%). To the best of our knowledge, this is the first work that examines the effect of software approximations on application resiliency.
- We further examine the SDCs caused by the approximate algorithms using a novel quality metric suitable to the domain of UAV image analytics. We show that most of the SDCs generated by the applied approximations have small quality degradations and can potentially be tolerated by the application.

In summary, we show that software approximation can be utilized to achieve significant gains in performance and energy without affecting application resiliency. This work does not claim to cover all possible types of approximations (or even the best ones) or comment on the general resilience of different techniques. Instead, the intent of the paper is to encourage a comprehensive evaluation (performance, power, resilience) of system optimizations and show that highly effective and resiliency-aware software approximations are possible. While we study a particular domain and report those results, the idea of holistically measuring system resiliency across different approximation knobs is generally applicable.

II. Background

A. Video Summarization

UAVs are increasingly being used to perform tasks such as surveillance of hostile targets and rescue and recovery in the event of natural disasters. For decisive action in all these scenarios, it is essential to first extract and summarize the concrete context from the input video stream captured by the moving UAV. This operation is termed as *Video Summarization*. A sophisticated video summarization work-flow requires application of several Computer Vision techniques.

One such example of an end-to-end application flow is described in Viguier *et al.* [3], where the UAV multimedia processing pipeline focuses on summarizing videos captured by the camera on-board the UAV. The framework for achieving this is shown in Figure 2. In order to achieve large data reduction without significant loss of events of interest, two types of summarizations are included: coverage and event summarization. Coverage summarization involves a complete panorama generation which describes the entire video with a single image that represents the entire spatial coverage area of the camera. The coverage summarization relies on spatially relating different video frames from the cameras, projecting them into a common view space, and stitching them together to build a single panorama. Event summarization comprises of tasks such as detection, recognition and tracking of moving objects such as vehicles and pedestrians. Finally, both intermediate results are integrated by overlaying the tracks (of

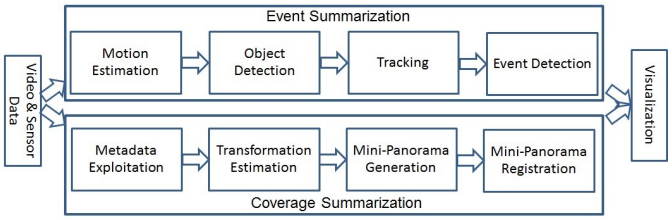


Fig. 2: Video summarization for UAV videos.

moving objects) on the panorama to create a comprehensive and concise summarization of a whole UAV video.

In this paper, we focus on *coverage summarization*. In particular, we focus on algorithms that perform the task of generating video panoramas of the landscape covered by the cameras on a UAV and on energy-efficient and reliable implementations of the same.

B. Approximate Computing

Approximate computing is a fast growing trend that allows controlled relaxation of correctness for better performance and energy. Users in these systems are typically willing to tradeoff some inaccuracies in the program output for other system benefits. Many techniques have been proposed that leverage approximate computing at the software [7], [8], [9], [10], [11], [12], programming language [13], [14], [15], [16], [17], [18], [19] and hardware [20], [21], [5], [22], [23], [24] level for improved performance, energy or reliability. Since we measure application level resilience in this work, we focus our analysis to software approximations. In particular, we study three broad classes of software approximations.

(1) Input sampling: In this class of approximation, computation is only performed over a subset of the input. This class of approximation is especially popular in big data analytics [25] where the amount of data over which the computation needs to be performed is prohibitive in time and resources.

(2) Selective Computation: Another popular class of approximations are those in which only a fraction of the work is performed compared to the precise program. While the underlying algorithm remains unchanged, selective computations are simply skipped or dropped [7].

(3) Algorithmic Transformation: These approximations transform the code and replace precise but expensive computation with cheap but imprecise computation.

The selection of which approximations to apply depends on the application/domain and the end goal. For example, approximations skipping certain loop iterations [7] and approximations dropping some synchronizations [9] in the program both belong to the broad category of selective computation and either, both or neither might be an appropriate approximation for a given application. In Section III we describe approximations belonging to each of the categories described above that are specific to the application we study.

III. Video Summarization Algorithm

As described in Section I, the system architecture of interest in this work is a model where a swarm of UAVs is engaged in image scanning, analysis and stitching with the end goal of creating a global panorama of the observed landscape. Towards this goal, we describe in this section a video (image) stitching algorithm that we have developed and implemented in our experimental evaluation platforms. Our application (henceforth referred to as the *Video Summarization (VS)* algorithm) takes input videos captured by moving cameras and generates panoramas that provide a global view of the landscape. Since the input video is a concatenation of images captured by (various) moving cameras, it can contain various segments with dissimilar viewing angles and settings. Each of these segments are summarized by *mini-panoramas* and are stitched into a global panorama (simply referred to as *panorama*) at a later stage. In this paper, our analysis is restricted to the generation of a panorama from video captured by a single camera on-board a single UAV.

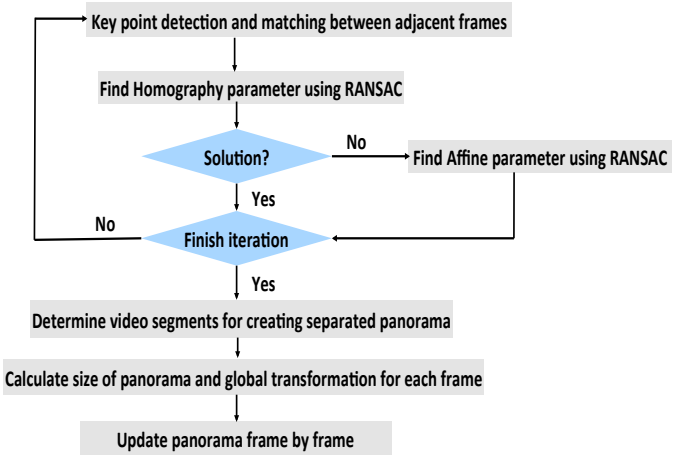


Fig. 3: Flowchart describing the key tasks that comprise the Video Summarization Algorithm.

A. Functional Overview

While a full detailed description of the algorithm is provided in [26], for the sake of brevity, we describe the key capabilities of the algorithm [27] in the following paragraphs. A representative flow of the algorithm is shown in Figure 3.

One of the fundamental functions performed by the *VS* algorithm is the comparison, transformation and stitching of two images from the input video. The algorithm first identifies key regions of interest (*key points*) within each image and then looks for matching key points within the images to identify potential common areas. It then applies transformations to the two images so that they are aligned correctly and have the same scale, lighting, perspective etc., before proceeding to stitch them together. Figure 4 shows this process using two sample images. We utilize FAST (Features from Accelerated Segment Test) detectors [28], [29] and ORB (Oriented FAST and Rotated BRIEF) descriptors [30] to achieve efficient and accurate feature point detection and matching. RANSAC (RANdom SAmple Consensus) [31] is used to compute the homography transformation between the two images.

Fig. 4: Simple example of stitching two images.

Using the technique described above, successive frames of the input are pair-wise compared in the initial pass of the algorithm. However, not every pair of adjacent frames has enough matching *key points* to compute the homography transformation. In this case, we estimate a simpler affine transformation which requires fewer matching points. If sufficient number of matching points cannot be found even for the affine transformation, the corresponding frame is discarded. To generate the overall output panorama we align every frame to the first by transforming all the frames to have the same coordinate system as the first frame by using the homography transformations described above.

There are various other sophisticated elements of the stitching algorithm that are used to improve the rendered quality of the output panorama. The mathematical details of the transformations and corrective actions (e.g., to avoid blurs and distortions) are omitted here for brevity. Depending on the quality and number of the input video clips (collected by the moving cameras), the amount of computation performed by the video summarization procedure can vary.

B. Inputs to the Video Summarization Algorithm

We evaluate the *VS* application using two aerial videos from the VIRAT (Video and Image Retrieval and Analysis Tool) dataset [32] – *09152008flight2tape1_2* (hereby referred to as Input 1) and *09152008flight2tape2_4* (hereby referred to as Input 2). We use an input size of 1000 frames for both inputs.

The VIRAT dataset was chosen for our evaluations to represent realistic scenarios of videos captured during aerial surveillance with variations in resolution, diversity in scenes, changes in scale, focus and camera angles. The two inputs that we profiled vary significantly in these aspects as well as in the nature with which these parameters vary in the video stream. For instance, the number of changes that occur in Input 1 are much higher than Input 2, leading to a much larger number of mini-panoramas generated in the first input set. These videos were sampled at periodic intervals to yield around 3000 frames across the duration of the entire video. In addition, we further downsampled the video by a factor of 3 to enable a statistically significant number of error injection experiments to run within a reasonable amount of time without perceivable loss in information or image quality.

IV. Approximate Video Summarization Algorithms

As described in Section III, the Video Summarization (*VS*) application is capable of effectively capturing several hours of video in single stitched image frames. However, given the constraints on power efficiency (of the on-board device) as well as real-time requirements of the mission, a complete and exact implementation of the algorithm may not be possible. In [4] and [33], the authors examine techniques to mitigate this limitation by dynamic adjustments of the link bandwidth and processor voltage/frequency.

In this paper, we consider software approximation, to the *VS* algorithm, as a means to realize performance and energy targets. Since computations involving images can be inherently tolerant to inaccuracies in data and/or compute, approximations to the the *VS* workflow have the potential to yield significant benefits without unduly compromising the quality of the final panorama image output. We study three different approximations (belonging to the three broad classes described in Section II-B and presented in the same order). The details of the approximate algorithms are described below:

(1) Random Frame Dropping (*VS_RFD*): In this algorithm, we randomly drop frames from the input stream. Apart from improving the effective frame rate, this input approximation aims to leverage redundancies in consecutive images captured by a moving camera without substantial degradation in output image quality. In this paper, we demonstrate results with up to 10% of the input frames being dropped.

(2) Key Point Down Sampling (*VS_KDS*): The *VS* algorithm described in Section III involves the computation of feature (key) points and attempts to match them across frames in order to be able to stitch the frames together. We propose an approximation in which we only perform matching on a fraction (one-third) of the key points as compared to the precise algorithm. This significantly reduces the computation time, which varies as $O(n^2)$ with the number of key points. In this algorithm, the source of error could be due to some frames being dropped on account of having insufficient matching key points. In such cases, it is hoped that the redundancy of the image will still enable us to obtain complete coverage of the input video in our summarized output.

(3) Simple Matching (*VS_SM*): In the default algorithm, each key point in the current frame is compared with all key points in the incoming frame and the two nearest neighbors are determined for each key point. The key point is included in the list of *good* matches only if the ratio of the distance between the nearest and 2nd nearest neighbor is above a certain threshold; i.e., the nearest match is sufficiently closer than the 2nd nearest. This reduces the probability of a false positive, i.e., the probability that the key point in a frame incorrectly maps to a point in the subsequent frame, even if, in reality, there is no matching object. In case of *VS_SM*, we alter the algorithm to determine only the single nearest neighbor for each key point. In addition, we place an upper bound on the actual distance value and consider only those matches whose nearest neighbor is within a fixed distance of the key point.

Hence, only those key points in the incoming frame which match almost perfectly with those in the original frame would be considered. Note that this technique still leaves room for some errors, for example, when there are two identical objects in the image. In such cases, both nearest neighbor distances could fall within the threshold and the mapping could happen to the incorrect object.

A. Effectiveness of the Approximate Implementations

An approximate algorithm has to produce acceptable quality outputs while enabling some system benefit (e.g. improved performance or energy efficiency). Hence, we examine the three approximate algorithms from the point of view of both system benefits and output quality to determine if they are good candidates for further study.

System benefits of approximation: We carried out an experimental evaluation of these algorithms on an IBM POWER-based server class machine. Figure 5 shows the Instructions Per Cycle (IPC), execution time and energy, normalized to the baseline *VS* algorithm. We observe that *VS_RFD* provides the maximum reduction in execution time (68%) for Input 1 by just dropping 10% of the total frames. On the other hand, *VS_KDS* yields the highest performance improvement of 18% in case of Input 2. Since the IPC (and hence, the power) remains relatively constant across the default and approximate implementations, the energy profile across the exact and approximate implementations varies similarly to that of the execution time.

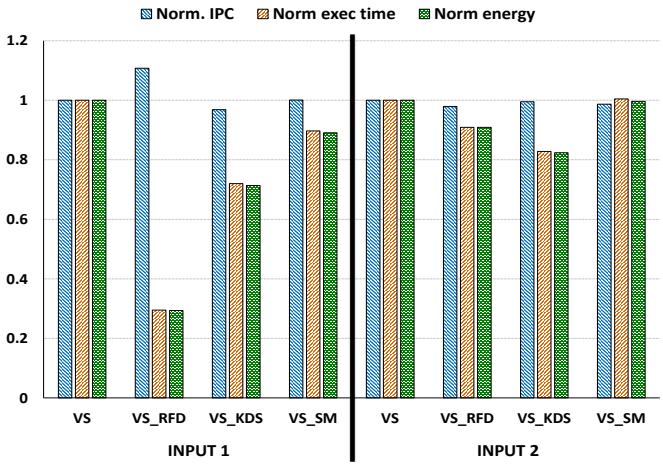


*Fig. 5: Comparison of IPC, execution time and energy of the proposed approximate algorithms (*VS_RFD, VS_KDS, VS_SM*) for Input 1 and Input 2, with the values normalized to the corresponding baseline (*VS*) for each respective input.*

Comparison of output quality: Figure 6 compares the output images generated by the baseline *VS* algorithm and the three approximations described for the two inputs. Visual inspection shows that the approximate algorithms generate output images of acceptable quality. Even in the approximate output image with the worst quality (*VS_RFD* for Input 1), the quality degradation is due to image perspective and all the pertinent information in the final panorama is retained.

Tradeoffs between performance and output quality: The difference between the two inputs is evident from the tradeoffs between performance and output image quality for each approximation. For instance, a visual inspection of the output panoramas generated by the baseline *VS* algorithm and the three approximations (Figure 6) show that Input 2 is more robust to the proposed approximation techniques as compared to Input 1. On the other hand, the performance benefits due to approximation are clearly greater in case of Input 1.

This difference in the impact of approximation on the two inputs can be attributed to the fact that the variation between consecutive frames is much more pronounced in Input 1 than in Input 2. The execution time improvement is primarily due to the polynomial complexity of the algorithm in terms of number of frames that are processed. In addition to the frames dropped by the approximate implementation, the algorithm also discards additional frames without stitching them to the overall panorama, when sufficient matching points are not found. Consequently, the proposed approximation techniques result in several frames of Input 1 being discarded. While this reduces the number of computations, resulting in greater performance and energy benefits as compared to Input 2, it also adversely affects the output quality to a greater extent. The differences between the output images can be further analyzed quantitatively by means of our proposed metric, described in further detail in Section V-D.

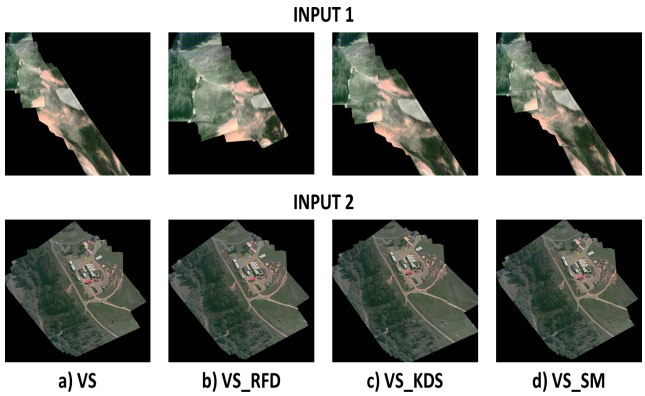


Fig. 6: Comparison of the output panoramas obtained from baseline VS algorithm (a) and various approximation techniques (b,c,d) for the two input image sets.

V. Methodology

In this section we describe the design methodology and the evaluation environment used to measure the resiliency of the *VS* algorithm as well as its approximate versions. In section V-C we describe a small case-study to understand the trade-offs of performing resiliency analysis on a full end-to-end application (such as the *VS* algorithm) vs. constituent small kernels. In section V-D, we define a metric to calculate the quality of the corrupted output produced by the application when perturbed by errors. We later use this metric to analyze the quality of the corrupted outputs produced by the approximate *VS* algorithms.

A. Measuring Resiliency of Video Summarization Algorithm

Error injection is a widely used error analysis technique where an error is injected (typically one at a time) in a real or simulated machine and the outcome (impact of the error) is studied [34], [35], [36], [37], [38], [39]. Our goal is to evaluate the application-level resiliency of the *VS* algorithm and its different approximate versions in the presence of hardware transient errors. The error model studied in this work assumes the occurrence of single bit errors in the architectural register file. The impact of an error on a program can be described by the following four outcomes:

(1) Mask: The error is masked by consecutive execution such that the application produces the correct output. This can happen if the error affects dead state or if the corrupted state is overwritten before being used.

(2) Crash: The error catastrophically affects the program state and results in the program crashing. For example, an error that leads to an out of bounds memory access.

(3) Silent Data Corruption (SDC): The error propagates through the program execution and corrupts the output. This is called a Silent Data Corruption because there is no obvious symptom of the error till the execution completes and the output is found to be corrupted.

(4) Hang: The error corrupts the internal state of the program such that neither completes nor crashes but hangs.

Comprehensively injecting errors in each potential error site in the program execution is prohibitively time consuming. For instance, in our study, each bit in every architectural register at every single execution cycle is a potential candidate for error injection. For most applications, the number of error sites is prohibitively large. Hence, we rely on statistical error injection in randomly selected error sites in the execution. This technique provides statistical summaries of the impact of errors on the application by estimating average rates for Mask, Crash, SDC and Hangs. Alternate, more comprehensive and higher precision techniques such as Relyzer [40] could be applied but are left to future work.

For accurately estimating the application resiliency, it is essential to perform error injections in a significant number of error sites that are uniformly distributed over the program execution. We use the term *error-site coverage* (or simply *coverage*) to indicate the relative robustness in the number and distribution of error sites picked for error injections.

We estimate the minimum number of error injection experiments needed to get an adequate statistical sample by observing the different rates of Mask, Crash, SDC and Hang over many error injections and the point at which these rates stabilize. In other words, the minimum number of error injections required are at the *knee* of the trend curves for the Mask, Crash, SDC and Hang rates. Beyond the *knee* of the curves, increasing the number of error injections should only change the outcome rates trivially.

B. Error Injection Environment

Error injection experiments are conducted on the IBM POWER-based machine, running Linux RHEL 6.5 operating system. We use the Application Fault Injection (AFI) [4] tool to perform error injections and evaluate the application's resiliency. Figure 7 shows the main components of AFI.

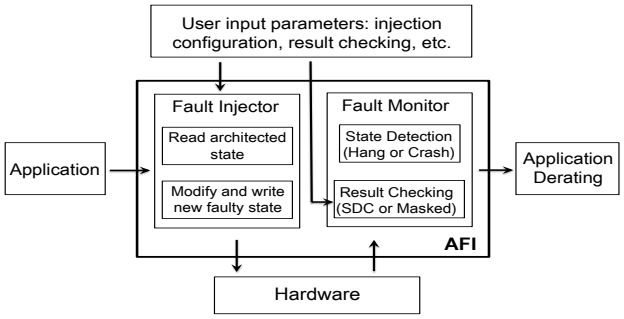


Fig. 7: Overview of the Application Fault Injection framework.

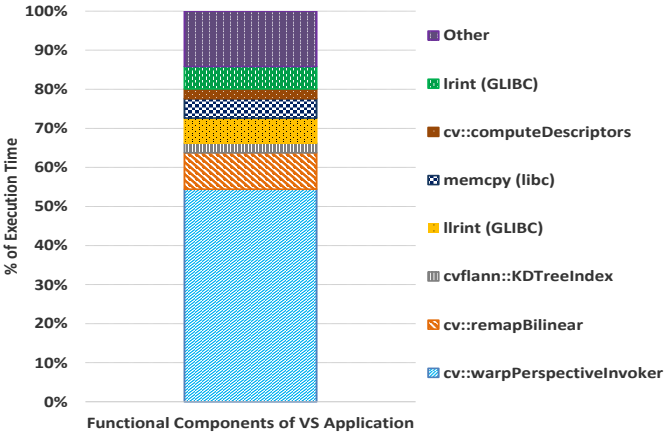


Fig. 8: Execution profile of the VS application

AFI is composed of two modules. The first module, *Fault Injector* takes the un-modified application binary, and injects error bits into the application's architectural state. Users can change the injection configurations to specify where to inject errors and how many errors to inject per run. For our study, we configure AFI to inject one single bit error (bit flip) in a general purpose register (GPR) or a floating point register (FPR). The execution cycle at which the error is injected is random. Once the program execution is continued after the error injection, the second module, the *Fault Monitor*, will check the application's running state and capture a potential hang or crash. If the application finishes normally, the Fault Monitor invokes a result checking procedure to determine if the outcome of the error injection is an SDC or Masked result.

We perform separate experiments for error injections in GPRs and FPRs as we want to separately examine the vulnerability of these two register types to single bit flips.

C. Studying full end-to-end workflow vs. small (hot) kernels

An optimized statically compiled binary (using GCC 4.8.2 and OpenCV version 2.4.9) of the *VS* algorithm spans ∼1.5 million lines of assembly instructions. Error injection experi-

ments on a large application like the *VS* algorithm, that run to completion, are time consuming and therefore limit the number of error injections that can be performed.

Since the application is a composition of many program and library functions, the question arises– can we simply carry out a resiliency study of some representative *hot* functions (functions that account for a significant fraction of the application execution time) and use the results to reason about the resiliency of the *VS* algorithm? If so, can we then study the resiliency of just those functions? This can lead to either reduced overhead (less number of error injections) or increased coverage (error injections take lesser time to run to completion and hence we can potentially do more of them).

To investigate further, we undertake a case study to perform resiliency analysis on a hot kernel taken from the *VS* application to see if the resiliency profile of the kernel is representative of the full end-to-end application. We show in Section VI-C that this is not the case and the result of such an analysis is sub-optimal. This further motivates the need to develop and analyze full end-to-end applications that realistically simulate the full workflow, as opposed to studying just small kernel benchmarks that perform individual tasks.

Fig 8 shows the execution time distribution (by function) for the *VS* algorithm extracted using the Linux utility tool *Perf* [41]. Approximately 68% of the execution time is spent in OpenCV libraries [42]. 54.4% of the total execution time is consumed by just one OpenCV function – *WarpPerspectiveInvoker*, which is called from the *WarpPerspective* function, that applies a perspective transformation to an image according to a transformation matrix. Thus, we choose *WarpPerspective* as the *hot* function whose resiliency profile we study.

We design a toy benchmark called *WP* that takes an image and a matrix as inputs and calls the OpenCV function *WarpPerspective* on them and returns the transformed image as the output. Essentially, *WP* is equivalent to having a standalone *WarpPerspective* function and the output of *WP* is the return value of the function as seen by the *VS* application. The function *WarpPerspective* in turn calls two other functions: *warpPerspectiveInvoker* and *remapBilinear*. We study the outcomes from error injections in GPRs in these two functions for *VS* and *WP*. Our error injection framework, AFI, gives us the ability to control where the errors are injected and for this experiment we only consider the error injection experiments that inject errors in the functions of interest and observe the outcome at the end of the program (either *VS* or *WP*).

D. Defining SDC quality

As described in Section V-A, we define any deviation in the application output due to an error as a Silent Data Corruption or SDC. SDCs are the least desirable outcome of errors since they are very hard to detect until the application execution is completed and the corrupted output is generated. At that time it is too late for recovery techniques to correct the error. Crashes, on the other hand, can be detected using low cost symptom-based detectors [35] and hence protecting error sites that produce crashes incurs low overhead. Since SDCs do not produce any easily detectable software symptoms, protection against SDCs is normally done through techniques like redundancy that have high overhead. In order to reduce the resiliency overhead, we aim to quantify the *egregiousness* or *severity* of the SDCs produced so we can identify tolerable or benign SDC error sites that do not need to be protected.

Since the *VS* application produces an image (*mini panorama*) as the output, the check to determine if an SDC was produced is an image comparison between the error-free application's output (henceforth referred to as the *golden output*) and the corrupted output produced by the application execution injected with an error (henceforth referred to as the *faulty output*). To determine if there is an SDC, AFI's result checking procedure simply compares the error-free output, known *a priori*, with the output produced by the erroneous execution, and classifies the outcome as an SDC if there is any difference between the two images.

In addition to knowing how many error injection experiments result in SDCs, we are also interested in quantifying the quality of the SDCs produced; i.e., the deviation between the golden and the faulty output. To do this, we define a quality metric which is calculated as follows:

Given a golden image *g_img* and a faulty image *f_img*, we first apply some global transformations to ensure that differences due to perspective, lighting, camera angle etc. are removed. We do this because, in our system, the end purpose is to use the output image of the *VS* application for identification, tracking and/or surveillance. Hence we are more concerned with the content of the image and can tolerate minor cosmetic disturbances in the final image. The two transformed image matrices obtained after this corrective step are *g_img_tr* and *f_img_tr*. The pixel by pixel difference of these two images is given by the matrix *pixel_diff_img*, such that

$$pixel\_diff\_img = g\_img\_tr - f\_img\_tr$$

Since in our scenario of interest, the final panorama is going to be viewed by a human being, we can tolerate some errors in the color gradation of individual pixels. Thus, we only wish to capture those differences in the image where the pixel coloration is significantly modified. For this purpose, we define another matrix *pixel_128_diff_img* where we only store values from *pixel_diff_img* if the difference value is greater than 128, that is over half the range for an 8 bit pixel which can assume values between 0 and 255. Then, the *relative_l2_norm*, which estimates the deviation of the faulty output image from the golden output image (in percentage) is described as,

$$relative\_l2\_norm = \frac{||pixel\_128\_diff\_img||_2}{||g\_img\_tr||_2} * 100$$

where, for an image X having n pixels $x_1, x_2, \ldots, x_n$

$$||X||_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$$

Once the *relative_l2_norm* of a faulty image has been calculated, we then assign that SDC an integer number called the *Egregiousness Degree (ED)* which corresponds to the floor of its *relative_l2_norm* value. The higher the ED, the worst the

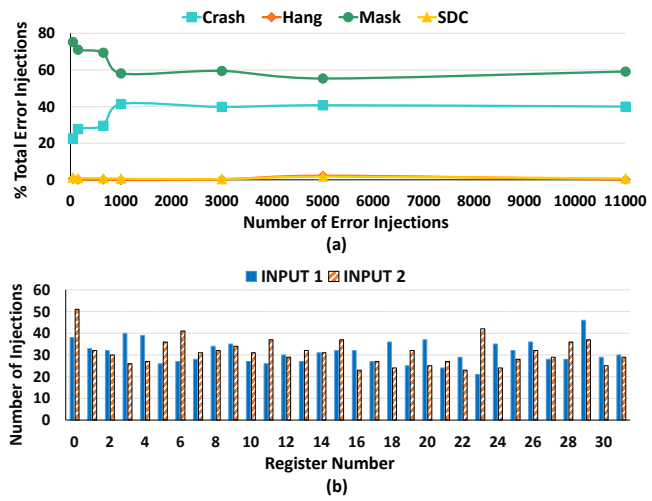


Fig. 9: Graphs to show Coverage of error injection experiments. (a) Different error injection outcome rates with increasing number of error injection experiments for VS algorithm. The knee of the curve stabilizes at 1000 error injections. (b) Number of errors injected in different GPRs across 1000 experiments show a uniform distribution.

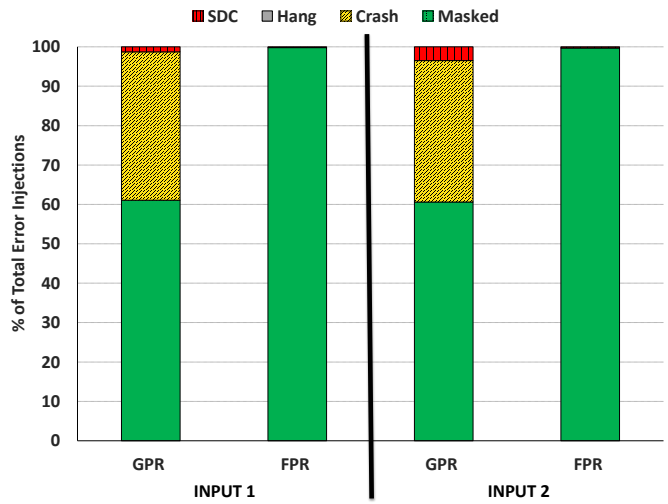


Fig. 10: Resiliency Profile for the VS algorithm. We show the different error outcome rates for errors injected in GPR and FPR registers for the two different inputs.

quality of the corrupted output image. For example, if an SDC output has a *relative_l2_norm* of 10.25%, it is assigned an ED of 10. Any SDC that has a *relative_l2_norm* of greater than 100%, is not assigned an ED and is automatically categorized as an egregious SDC that must be protected.

VI. Results

A. Resiliency Profile of Video Summarization Algorithm

As described in Section V-A, we determine the minimum number of error injections needed by studying the trend curves of the Mask, Crash, SDC and Hang rates with increasing number of error injections. As seen in Fig 9(a), the trend curves for the different rates start stabilizing after 1000 error injections and only vary slightly with increasing error injections. Thus, we conclude that a minimum of 1000 error injections is required to provide a statistical summary of the *VS* algorithm. Unless otherwise specified, all our experiments use 1000 error injections (for each type of register; combined GPR and FPR is 2000 error injections).

The random error injections also provide good coverage in terms of the registers and bits in which the errors are injected. A representative histogram is presented in Fig 9(b). It shows that the errors are uniformly distributed among the 32 GPRs (for both inputs). We similarly confirm that the errors are uniformly distributed among 64 bits within the registers. For brevity, we have shown the coverage data (minimum error injections required and register coverage) for error injections in GPRs. Error injection experiments for all the different algorithms (GPR and FPR) show similar trends.

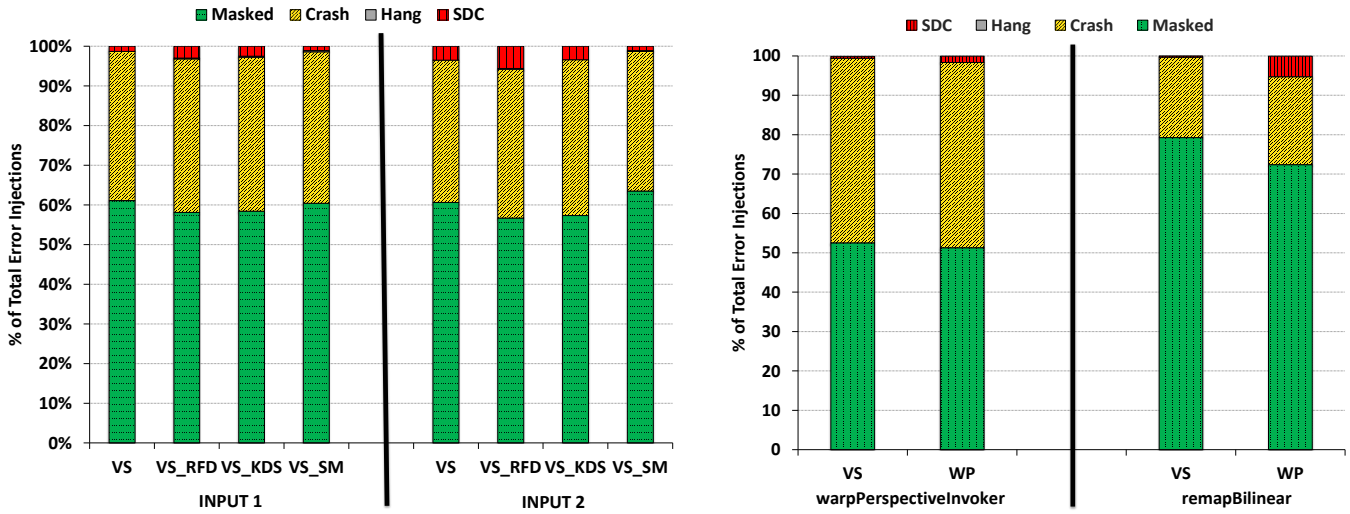
Figure 10 shows the different error injection outcome rates for 1000 error injections each in GPRs and FPRs for the *VS* algorithm. The resiliency profile looks very different for injections in the GPR and the FPR registers and we will explore these differences in the following paragraphs.

Error Injections in GPRs: Instructions that use GPRs form the bulk of our application and are heavily used in memory and control instructions and hence errors in them lead to the large Crash rate (40.16%). Analyzing the Crash outcomes further, we see that the majority of the crashes can be attributed to the following two causes: 1) Segmentation Faults that generally occur due to memory access violations (92%), and 2) Abort signals raised by the application/library when it encounters internal constraint violations (8%). Analyzing the Crash causing error sites further, we see no clear trend that corruption of certain registers or bit positions in the registers are more likely to result in a Crash. This is primarily because all the GPR registers are used heavily in control (corruption of any bit can cause a Crash) and memory (higher order bits more likely to cause a Crash) operations and, hence, are vulnerable to catastrophic outcomes when corrupted.

Error Injections in FPRs: Errors injected in the FPRs of the *VS* application are Masked 99.7% of the time. This is due to the way FPRs are used in the application. The *VS* algorithm operates on images which are stored as 8-bit integer pixels. Floating point operations are only used when some manipulation of the pixels is required. To do this, the integer pixels are converted to floating point, some transformation is applied and then they are converted back to integer using a saturation algorithm. The saturation algorithm causes many potentially SDC causing errors to become masked.

B. Resilience of Approximate VS Algorithms

Figure 11a shows the error injection results for 1000 error injection experiments in the GPRs of the different approximation algorithms compared with the baseline *VS* application for both the inputs. Similar to the baseline *VS* algorithm, FPR error injections in the approximate algorithms are masked $> 99.5\%$ of the time and hence we do not show them here. The Crash, Mask and Hang rates of the approximate algorithms is very



(a) (b)

Fig. 11: Resiliency profiles (Crash, SDC, Mask and Hang rates) for the different VS algorithms and toy benchmark WP. (a) Resiliency Profile for the VS algorithm and its approximate versions. We show the different error outcome rates for errors injected in GPR register for the two different inputs. (b) Comparison of the Masked, SDC and Crash rates for error injections in two hot functions for VS application and the stand-alone toy application WP.

similar to the baseline *VS* algorithm. This is not surprising since the execution profiles of the approximate algorithms are very similar to the baseline *VS* algorithm. For Input1, the SDC rates increase from 1% (*VS*) to 3% and 2.5% for *VS_RFD* and *VS_KDS* respectively. In both these approximate algorithms, the errors that may have been masked in the final image (due to overlap by similar frames in the stitching process) are now exposed as SDC due to a reduction in redundancy; precipitated by dropping frames from input in *VS_RFD* or due to insufficient matching key-points in *VS_KDS*.

C. Trade-offs of studying an end-to-end workflow

As discussed in Section V-C, we ask the following question: can we estimate the resiliency of the *VS* application by studying the resiliency of the representative stand-alone *WP* application? The results of the error injection experiments for both *VS* and *WP* are shown in Figure 9(b).

The Crash, Mask and SDC profiles of the standalone *WP* is different from that of an end-to-end workflow like the *VS*. In the *VS* application, the output of the *WarpPerspective* function would then be used to perform some other computation further down the workflow and, hence, there is a compositional effect where multiple computations flow into each other. This causes the effects of an error to manifest differently than if the workflow ended at the output of the *hot* function. In our case, the compositional effect leads to higher masking as the SDCs that are generated by errors in the *WarpPerspective* function are masked later in the workflow (for example, an adjacent image could later be stitched over the area corrupted by the function output). Hence, we conclude that it is essential to analyze an entire end-to-end workflow, instead of just studying *hot* kernels/functions, to get an accurate understanding about the resiliency behavior of an application.

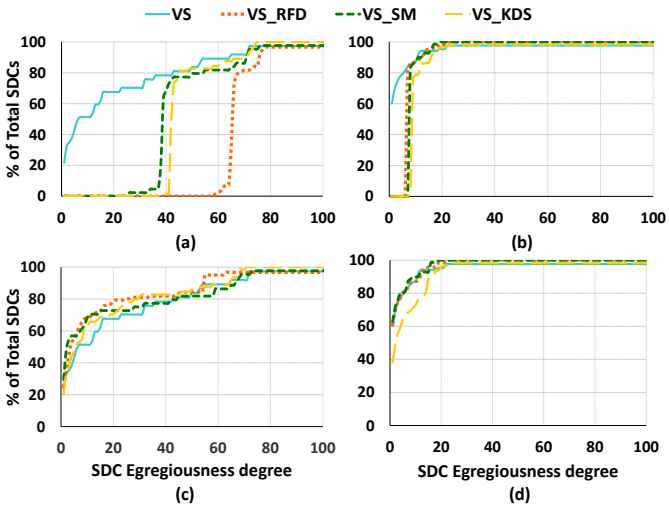


Fig. 12: Quality of SDCs generated by GPR error injections in different video summarization algorithms. Each point on a given curve represents the percentage of SDCs (Y axis) generated that have an ED less than or equal to the ED represented on the X axis. (a) and (b) - The ED of the SDC is calculated by comparing against VS_golden for Input1 and Input2 respectively. (c) and (d) - The ED of the SDC is calculated by comparing against the corresponding Approx_golden for Input1 and Input2 respectively. Some of the curves do not reach 100% on the Y axis due to a very small fraction of SDCs that are classified as needing protection and not assigned an ED.

D. SDC quality

Figure 12 classifies SDCs according to their ED. In order to study a statistically significant number of SDCs, we perform 5000 GPR error injections per input and analyze the resulting SDCs. To calculate the ED of an SDC output image, we compare it to a baseline *golden* image. For the SDCs produced by

the *VS* algorithm, this is straightforward as the golden image is the output of the error-free execution of the *VS* algorithm. For SDCs produced by the approximate algorithms, there are two potential golden images to compare against – the golden *VS* output (*VS_golden*) or the golden output of the corresponding approximate algorithm (*Approx_golden*). For example, ED of an SDC produced by error injection in *VS_RFD* can be calculated by either comparing it to *VS_golden* or by comparing it to *VS_RFD_golden*. We show the distribution of SDC egregiousness using both these methods.

Figure 12(a) and Figure 12(b) show the ED of the SDCs when compared against *VS_golden* for Input1 and Input2 respectively. The degradation in SDC quality in the approximate algorithms, as evidenced by the larger fraction of SDCs having higher EDs, is particularly sharp for Input1 (Figure 12(a)). On further analysis we observe that this is because the deviation between *Approx_golden* and *VS_golden* calculated using the metric specified in Section V-D is large. Even though we verified by visual inspection that the approximate algorithm outputs were acceptable (Section IV-A), our metric assigns them a large ED. This may imply that our metric is very conservative and we undertake a discussion about this in Section VII. For example, the ED of the *VS_SM_golden* for Input 1 when compared to *VS_golden* is 37. It thus follows that all subsequent SDCs produced by *VS_SM* will have an ED greater than or equal to 37. This is the reason we see the shift in the ED curves of *VS_SM* with respect to the baseline *VS* in Figure 12(a).

Thus, to get a true understanding of the quality of the SDCs produced by the approximate algorithms, we estimate their egregiousness by comparing them to their corresponding *Approx_golden* output (Figure 12(c) and Figure 12(d)). The graphs show that the overall trend for the SDC quality for the *VS* and its approximate algorithms are very similar. The approximations do not fundamentally change the quality of the SDCs produced. For Input 2 (Figure 12(d)), the SDCs from *VS_KDS* have slightly worse quality (80% of SDCs produced by *VS* have ED less than 6 as opposed to ED of 14 for *VS_KDS*). This follows the trend seen in Section IV-A, where for Input 2, *VS_KDS* shows the most energy gains from less computation as a result of dropped frames. This in turn leads to a degradation of output quality. Another trend seen is that overall, the SDCs produced are relatively benign (even with our conservative metric). For example, for Input 2, 87%, 87%, 90% and 73% of the SDCs for *VS*, *VS_RFD*,*VS_SM* and *VS_KDS* respectively have an ED of less than 10. Thus, a large majority of the SDC causing error-sites need not be protected if an error of 10% is acceptable.

Hence, although approximating the *VS* application minimally changes its resiliency profile by slightly increasing the number of SDCs generated, this is offset by the fact that a large percentage of these SDCs may be tolerable and hence the cost of protecting them is low. Thus, it is possible to realize safe, yet efficient approximations for this state of the art Video Summarization algorithm from the point of view of performance, power and reliability.

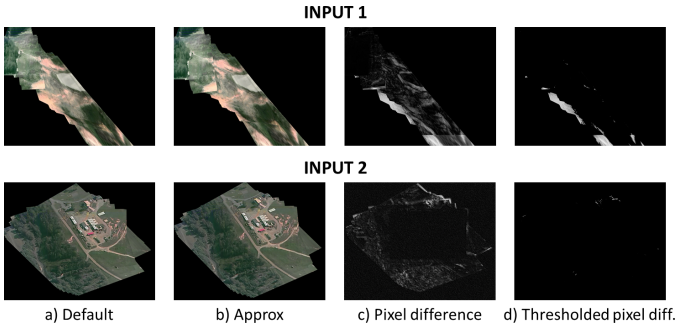


*Fig. 13: a) Default output (*VS*) b) Approximate output (*VS_SM*) c) Absolute pixel difference between default and approximate outputs d) Thresholded difference between default and approximate outputs.*

VII. Discussion on SDC Quality Metric

Gauging if an approximate algorithm is good enough or if an SDC is tolerable in image processing applications like the *VS* algorithm is heavily dependent on the image comparison algorithm that calculates how closely the approximated image or the faulty image matches the original image. While manual inspection is still the best way to determine if the quality of an image is acceptable, this is impractical in cases where a large number of such images are generated or when an automated decision has to be made based on the error seen in the output. In Section V-D, we outline an algorithm and metric to estimate the error in the output image, but our algorithm can produce false positives and can label some SDCs as more egregious than they actually are. For the outputs of the *VS_SM* algorithm the *relative_l2_norm* generated by the image comparison algorithm is approximately 37% and 8% for Input1 and Input2 respectively. This is because as can be seen in Figure 13(c), the pixel difference of the two images is considerable as the pixels in the faulty image have slightly shifted when compared to the default image. But to a human viewing these two images, there is no perceivable difference. Another factor to consider is that two images having the same *relative_l2_norm* may not be equally egregious depending on the final usage of the output. For example, even if 30% of a faulty image is blacked out, it may still be useful for surveillance or tracking if the remaining 70% had useful information that can be deciphered by a human being. Estimating an automated metric to compare images used for such domains remains an open problem.

VIII. Related work

A. Approximate computing

Section II-B discusses many trends in approximate computing. A related area is analysis that performs criticality testing [43], [44] and works that take advantage of soft computations - resilient code regions that result in tolerable output corruptions, when perturbed by errors - to reduce resiliency overheads in approximate environments [45], [46], [47], [48]. To the best of our knowledge, this is the first work that directly measures the resilience of approximate algorithms.

B. Resiliency

Soft error resilience has been studied over several decades (dating back to 1978 [49]) with a large body of more recent work at various levels of design hierarchy, such as the logic level [50], [51], microarchitectural level [52], [53], and architectural level [54], [55], [56], [57]. For example, Mukherjee *et al.* proposed the concept of the Architectural Vulnerability Factor (AVF) in [54] to quantify the resilience of various architectural components. In [52], Kim *et al.* studied the soft error sensitivity of functional blocks using software simulated fault injections into the RTL model of a microprocessor (picoJava-II). Though showing significant masking effects of more than 85% reported by Wang *et al.* in [53], none of these prior works specifically considered the algorithmic effects on soft error resilience, nor did they evaluate approximation mechanisms with acceptable end-quality.

More recently, there has been an increasing interest in studying resilience at the application level for low-cost reliability solutions. SWAT/mSWAT [58], [59], [60], [61] take advantage of application's abnormal behavior, referred to as *symptoms*, to identify faulty units using light-weight diagnosis algorithms. In [62] Thomas *et al.* propose the term *EDC* describing outcomes that deviate significantly from the error-free outcomes of an application. Based on heuristics learned from EDC characterization, they propose a detection mechanism to identify variables and locations to protect against EDC. We propose a novel quantitative metric for EDC evaluation on a complete video stitching algorithm and approximation algorithms to achieve improvement in energy efficiency and performance without significant loss in end-quality.

C. Computer vision for UAV-based mobile cognition

Extensive research has gone into image-stitching algorithms in the field of computer vision. In [63], Szeliski describes various algorithms for aligning and stitching images into seamless 2D photo-mosaics. Various state-of-the-art algorithms to handle and summarize video content captured on-board a UAV-based processor, have been described in [27]. Rane et al. [64] proposed a method to evaluate mosaic quality using maximum information retrieval. The method uses the similarity between the stripes of the mosaic and the original frames to evaluate performance of mosaicking methods. The videos in the VIRAT dataset contain both translational and rotational movements. However, unlike the algorithm evaluated in this paper, this method works when camera has only translational movement. The evaluation method proposed by Camargo *et al.* [65] uses the distances between the corresponding keypoints in all frames after the mosaic is generated. This method is used to compare different optimization methods for parameter estimation, but does not consider the image distortions caused by fault injection. The paper [66] empirically evaluates the detectability of objects of interest for human observers when temporally local mosaics are applied on the live aerial video, but cannot provide quantitative evaluation for fault injection. Paalanen *et al.* [67] proposed a method to evaluate the mosaic quality using ground truth data. However, ground truth data can only be obtained in synthetic datasets. Since the dataset that we use for our evaluations is a real-word one, determining the ground truth is difficult. El-Saban et al. [68] use human eye to measure precision/recall of the mosaic quality of image pairs. However, this method too cannot provide scientific measurement of the distortion caused by fault injection.

IX. Conclusion

In this work we study an end-to-end video summarization *VS* application that serves as a representative emerging workload for the domain of Real Time Edge Computing. We characterize the workflow of the application and examine three different approximation techniques to improve the power and performance efficiency of the workload while maintaining sufficient output integrity. We undertake a detailed resiliency study of the application as well as its approximate versions and show that the approximations do not degrade the resiliency of the baseline algorithm. We further introduce metrics to quantify the error introduced in an output image and use them to understand the behavior of SDCs in the different Video Summarization algorithms. We show that many of the SDCs produced by the application can be tolerable to the end user and hence can reduce the cost of protecting the application against transient faults.

References

[1] G. Ananthanarayanan, P. Bahl, P. Bodk, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *IEEE Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[2] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, pp. 30–39, Jan 2017.

[3] R. Viguier *et al.*, "Resilient mobile cognition: Algorithms, innovations, and architectures," in *ICCD*, 2015.

[4] L. Wang *et al.*, "Power-efficient embedded processing with resilience and real-time constraints," in *ISLPED*, 2015.

[5] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design.," in *ETS*, pp. 1–6, 2013.

[6] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," *SIGPLAN Not.*, 2012.

[7] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *SIGSOFT FSE*, pp. 124–134, 2011.

[8] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *Programming Language Design and Implementation*, PLDI, pp. 198–209, 2010.

[9] 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, pp. 13–24, 2013.

[10] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications," *Multimedia, IEEE Transactions on*, vol. 15, no. 2, pp. 279–290, 2013.

[11] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and H. Esmaeilzadeh, "Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration," in *International Symposium on Computer Architecture*, ISCA, 2016.

[12] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali, "Proactive control of approximate programs," in *Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pp. 607–621, 2016.

[13] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, (New York, NY, USA), pp. 33–52, ACM, 2013.

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

[15] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris, "Flexjava: Language support for safe and modular approximate programming," in *Joint Meeting on Foundations of Software Engineering*, 2015.

[16] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "Accept: A programmer-guided compiler framework for practical approximate computing," in *Technical Report UW-CSE-15-01-01, University of Washington*, 2015.

[17] 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, pp. 164–174, 2011.

[18] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability type inference for flexible approximate programming," in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, pp. 470–487, 2015.

[19] J. Park, X. Zhang, K. Ni, H. Esmaeilzadeh, and M. Naik, "Expax: A framework for automating approximate programming," in *Technical Report, Georgia Institute of Technology*, 2014.

[20] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Designing a processor from the ground up to allow voltage/reliability tradeoffs," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1–11, 2010.

[21] J. Sartori and R. Kumar, "Architecting processors to allow voltage/reliability tradeoffs," in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES, pp. 115–124, 2011.

[22] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Microarchitecture (MICRO), International Symposium on*, pp. 449–460, 2012.

[23] J. San Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, "Doppelganger: A cache for approximate computing," in *International Symposium on Microarchitecture (MICRO)*, 2015.

[24] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," *SIGPLAN Not.*, vol. 46, no. 3, pp. 213–224, 2011.

[25] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. I. Jordan, S. Madden, B. Mozafari, and I. Stoica, "Knowing when you're wrong: building fast and reliable approximate query processing systems," in *International Conference on Management of Data, SIGMOD*, pp. 481–492, 2014.

[26] K. Swaminathan *et al.*, "A case for approximate computing in real-time mobile cognition," in *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2015.

[27] C. Lin *et al.*, "Moving camera analytics: Emerging scenarios, challenges, and applications," *IBM JRD*, 2015.

[28] E. Rosten and T. Drummond, "Fusing points and lines for high performance tracking.," in *ICCV*, 2005.

[29] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *ECCV*, 2006.

[30] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: an efficient alternative to sift or surf," in *ICCV*, 2011.

[31] M. Fischler and R. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, 1981.

[32] S. Oh *et al.*, "A large-scale benchmark dataset for event recognition in surveillance video," in *CVPR*, 2011.

[33] A. Vega *et al.*, "Resilient, UAV-embedded real-time computing," in *ICCD*, 2015.

[34] A. Meixner, M. Bauer, and D. Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," in *MICRO*, 2007.

[35] M.-L. Li *et al.*, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design," in *ASPLOS*, 2008.

[36] M.-L. Li *et al.*, "Accurate Microarchitecture-Level Fault Modeling for Studying Hardware Faults," in *HPCA*, 2009.

[37] A. Pellegrini *et al.*, "CrashTest: A Fast High-Fidelity FPGA-based Resiliency Analysis Framework," in *ICCD*, 2008.

[38] A. Pellegrini *et al.*, "CrashTest'ing SWAT: Accurate, Gate-Level Evaluation of Symptom-Based Resiliency Solutions," in *DATE*, 2012.

[39] S. Nomura *et al.*, "Sampling + DMR: Practical and Low-overhead Permanent Fault Detection," in *ISCA*, 2011.

[40] S. Hari, S. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting Application-level Fault Equivalence to Analyze Application Resiliency to Transient Faults," in *ASPLOS*, 2012.

[41] "perf: Linux profiling with performance counters."

[42] "Open source computer vision library (OpenCV)," 2015.

[43] M. Carbin and M. C. Rinard, "Automatically identifying critical input regions and code in applications," in *International Symposium on Software Testing and Analysis*, ISSTA, pp. 37–48, 2010.

[44] B. Nongpoh, R. Ray, S. Dutta, and A. Banerjee, "Autosense: A framework for automated sensitivity analysis of program data," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[45] Q. Shi, H. Hoffmann, and O. Khan, "A cross-layer multicore architecture to tradeoff program accuracy and resilience overheads," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 85–89, 2015.

[46] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *International Symposium on Microarchitecture (MICRO)*, pp. 1–14, 2016.

[47] P. Roy, R. Ray, C. Wang, and W. F. Wong, "Asac: Automatic sensitivity analysis for approximate computing," in *Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pp. 95–104, 2014.

[48] A. Thomas and K. Pattabiraman, "Error detector placement for soft computation," in *Dependable Systems and Networks (DSN)*, pp. 1–12, 2013.

[49] J. Ziegler *et al.*, "IBM Experiments in Soft Fails in Computer Electronics (1978 - 1994)," *IBM JRD*, 1996.

[50] E. Czeck and D. Siewiorek, "Effects of Transient Gate-Level Faults on Program Behavior," in *Int. Symp. on Fault-Tolerant Computing*, 1990.

[51] P. Shivakumar *et al.*, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," in *DSN*, 2002.

[52] S. Kim and A. Somani, "Soft Error Sensitivity Characterization for Microprocessor Dependability Enhancement Strategy," in *DSN*, 2002.

[53] N. Wang, J. Quek, T. Rafacz, and S. Patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," in *DSN*, 2004.

[54] S. Mukherjee *et al.*, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *MICRO*, 2003.

[55] A. Biswas *et al.*, "Computing Architectural Vulnerability Factors for Address-Based Structures," in *ISCA*, 2005.

[56] X. Li, S. Adve, P. Bose, and J. Rivers, "SoftArch: An Architecture-Level Tool for Modeling and Analyzing Soft Errors," in *DSN*, 2005.

[57] X. Li, S. Adve, P. Bose, and J. Rivers, "Architecture-Level Soft Error Analysis: Examining the Limits of Common Assumptions," in *DSN*, 2007.

[58] M.-L. Li *et al.*, "Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults," in *DSN*, 2008.

[59] M.-L. Li *et al.*, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *ASPLOS*, 2008.

[60] S. Sahoo *et al.*, "Using Likely Program Invariants to Detect Hardware Errors," in *DSN*, 2008.

[61] S. Hari *et al.*, "mSWAT: Low-Cost Hardware Fault Detection and Diagnosis for Multicore Systems," in *MICRO*, 2009.

[62] A. Thomas and K. Pattabiraman, "Error Detector Placement for Soft Computation," in *DSN*, 2013.

[63] R. Szeliski, "Image alignment and stitching: A tutorial," 2004.

[64] K. Rane *et al.*, "Mosaic evaluation: An efficient and robust method based on maximum information retrieval," *Int. J. Computer Applications*, 2013.

[65] A. Camargo, Q. He, and K. Palaniappan, "Performance evaluations for super-resolution mosaicing on UAS surveillance videos," *Int J Adv Robotic Systems*, 2013.

[66] B. Morse *et al.*, "Application and evaluation of spatio-temporal enhancement of live aerial video using temporally local mosaics," in *CVPR*, 2008.

[67] P. Paalanen, J.-K. Kämäräinen, and H. Kälviäinen, "Image based quantitative mosaic evaluation with artificial video," in *Image Analysis*, pp. 470–479, 2009.

[68] M. El-Saban, M. Izz, A. Kaheel, and M. Refaat, "Improved optimal seam selection blending for fast video stitching of videos captured from freely moving devices," in *ICIP*, 2011.