Managing the Performance/Error Tradeoff of Floating-point Intensive Applications

RAMY MEDHAT, University of Waterloo
MICHAEL O. LAM, James Madison University
BARRY L. ROUNTREE, Lawrence Livermore National Lab
BORZOO BONAKDARPOUR, McMaster University
SEBASTIAN FISCHMEISTER, University of Waterloo

Modern embedded systems are becoming more reliant on real-valued arithmetic as they employ mathematically complex vision algorithms and sensor signal processing. Double-precision floating point is the most commonly used precision in computer vision algorithm implementations. A single-precision floating point can provide a performance boost due to less memory transfers, less cache occupancy, and relatively faster mathematical operations on some architectures. However, adopting it can result in loss of accuracy. Identifying which parts of the program can run in single-precision floating point with low impact on error is a manual and tedious process. In this paper, we propose an automatic approach to identify parts of the program that have a low impact on error using shadow-value analysis. Our approach provides the user with a performance / error tradeoff, using which the user can decide how much accuracy can be sacrificed in return for performance improvement. We illustrate the impact of the approach using a well known implementation of Apriltag detection used in robotics vision. We demonstrate that an average 1.3x speedup can be achieved with no impact on tag detection, and a 1.7x speedup with only 4% false negatives.

CCS Concepts: •**Computer systems organization → Embedded software;** Robotics;

Additional Key Words and Phrases: Real-valued arithmetic, robotic vision, precision, performance, tradeoff

ACM Reference format:
Ramy Medhat, Michael O. Lam, Barry L. Rountree, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2017. Managing the Performance/Error Tradeoff of Floating-point Intensive Applications. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2017), 19 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Embedded systems are becoming more and more sophisticated as they utilize software that was traditionally only used on desktop machines. Computer vision, speech recognition, and machine learning modules are now becoming more common in embedded systems such as self-driving cars, IoT devices, and robots. One common feature of such applications is their extensive use of floating-point arithmetic to perform computationally intensive mathematics. When technology migrates to the embedded domain, it becomes even more crucial to optimize its performance and energy efficiency. Software that traditionally ran on powerful desktop machines may now be

This article was presented in the International Conference on Embedded Software 2017 and appears as part of the ESWEEK-TECS special issue.
 1539-9087/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

required to run on small, battery-powered embedded devices. In this context, heavy floating-point arithmetic can become a serious obstacle in developing resource-efficient embedded software.

Double precision is commonly used in mathematically complex algorithms since it provides the highest level of accuracy available in standard hardware. However, using single precision provides a performance boost due to less memory transfer, less cache occupancy, and fewer clock cycles for some mathematical operations. Better performance implies less energy consumption, especially with modern architectures optimizing idle power significantly. A single-precision implementation that is idle for a longer period of time is more energy efficient.

The advantages of single precision can potentially come at the cost of accuracy, especially in algorithms that require complex mathematics, such as algorithms involving computer vision, speech recognition, and AI. In this paper, we show the quantifiable impact of single-precision on the accuracy of a computer vision library [2]. In some cases, it is feasible to convert the entire code to single-precision and retain an acceptable level of accuracy. In fact, modern AI libraries provide the user with a switch to toggle between single and double precision [14]. However, as we show in this paper, the issue is often more subtle and complicated. To maximize performance while maintaining a level of accuracy that the developer can accept, the developer has to perform the tedious process of analyzing each part of their code to identify which variables or functions can be downgraded to single-precision without significant loss in accuracy. This involves an understanding of the algorithm, its input, and floating-point arithmetic.

To tackle the aforementioned subtleties, developers need an automatic approach to guide the process of precision downgrade without drilling into the details of the code. More specifically, developers need an automatic approach to identify which parts of the program are candidates for a downgrade from double to single precision. Candidates are parts of the program (variables, functions, instructions) which when downgraded result in an acceptable loss of accuracy. Such an automatic approach should aid the developer in managing the performance/error tradeoff, by providing information that can tell the user how much performance they can gain versus how much accuracy they can lose.

In this paper, we present a novel automatic approach for managing the performance/error tradeoff by identifying parts of the program that can tolerate lower precision with little increase in overall error. Our approach uses dynamic instrumentation to compute how a single-precision version of the program impacts error at the function granularity. We present methods to quantify and prioritize functions that can be converted to single precision. Then, we demonstrate our approach on a robotics vision case study, where robots detect 2D barcodes to identify locations and orientation. The proposed approach can present the developer with a spectrum of choices to manage the performance / error tradeoff. We present performance and error results of several points along that spectrum where we made manual changes to the code guided by the analysis results. In our case study, we demonstrate that we can achieve a speedup of 1.3x without any impact on the accuracy of detection. We also demonstrate that a speedup of up to 1.7x can be achieved with only a 4% drop in accuracy. We also study the energy and power consumption of the reduced precision implementations. We show that we can achieve a 16% energy reduction without sacrificing accuracy.

The paper presents the following contributions:

- An analysis tool that can trace the evolution of error per memory location / instruction, providing insight into how error changes as execution progresses. This insight can support sophisticated precision-switching mechanisms.
- An analysis tool that can isolate the error per function, canceling out the effect of error propagation and determining the isolated impact of a function's code on error.

- A method that provides developers with a quantifiable tradeoff between error and performance per function. This tradeoff aids the developer in making decisions regarding precision downgrade at the function level. Our tool provides recommendations as to which functions to downgrade iteratively. These recommendations are relatively simple to apply manually, since they are at the function level.

Organization. The rest of the paper is organized as follows: Section 2 describes the proposed approach. Section 3 describes the process of managing the performance / error tradeoff. Section 4 introduces the case study and the results of the analysis. Section 5 details our experiments with multiple precision levels and validates the quantification of performance and error provided at the analysis phase. Section 6 discusses our proposed analysis and the experimental results. Section 7 presents related work, and finally Section 8 presents the conclusion and future work.

2 PROPOSED APPROACH

2.1 Problem Statement

In this paper, we address the following problems:

- At function granularity, automatically quantify the error resulting from converting that function to use single precision floating-point.
- At function granularity, automatically estimate the performance benefit gained by converting that function to use single precision floating-point.
- Provide recommendations to the developer as to which mix of single and double precision functions to use to reach error and performance targets.

2.2 Evolution of Precision Error

For every memory location containing a floating-point variable in the original precision, shadow value analysis refers to maintaining a *shadow* value in the alternative precision. All mathematical computations on the original variables are repeated in the alternative precision on the shadow variables. This analysis produces error estimates that support decision-making regarding full or partial conversion of floating-point variables and code to the alternate precision.

In this paper, we build on top of an existing floating-point shadow value analysis tool [11] by extending it to support the proposed analysis. The tool uses Intel's Pin [13] to modify the target program via just-in-time instrumentation. To perform the analysis, our tool monitors memory access to double precision floating-point locations. It then maintains a map of memory addresses and shadow values, which can be in native single precision or any arbitrary precision. The tool detects SSE instructions and replicates them using the target precision. After the program under inspection terminates, the tool reports the relative error per memory location which is calculated as

$$\left| \frac{v_s - v_o}{v_o} \right|$$

where v_s is the shadow value and v_o is the original value.

In the remainder of this section, we describe the modifications we made to the tool in [11] to support the proposed analysis. Our objective is to study how error evolves as the program progresses with respect to both memory locations and program instructions. This analysis can help identify parts of the program that can be converted from double precision to single precision with an acceptable impact on error.

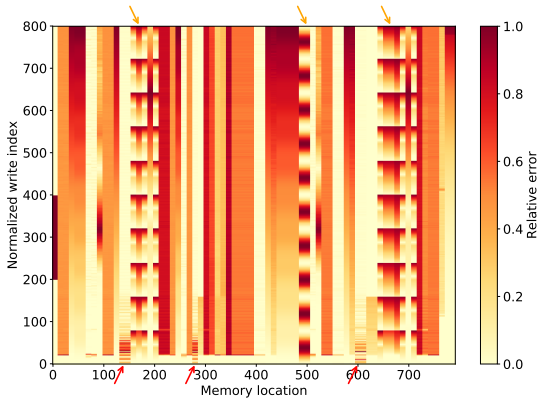

Fig. 1. Error trace per memory location. A darker pixel indicates higher error.

2.3 Tracing Memory Error

The first modification to the tool is to track the error at each memory write. This analysis can help us identify error behavior for floating-point variables. The following are possible scenarios:

- The error is consistently low, making the variable a good candidate for conversion to single precision.
- The error is consistently high, indicating that the variable should remain at the original double precision.
- The error increases as execution progresses. We have observed this behavior in scientific computations [7], where rounding error accumulates as the application proceeds. This indicates potential for migrating the variable to single precision mid-execution. We discuss this further in Section 8.
- The error decreases as execution progresses. We have observed this behavior in a laser beam stabilizing control system. In this system, the initial state exhibits large physical error. This reflects in a larger relative error as the control system tries to stabilize the laser on the target. Once the laser is stable, the error drops, even during the rhythmic disturbance caused by a motor. This scenario makes the case for running at single precision only after the controller has stabilized. This is also further discussed in the future work section.

To avoid large CPU and memory overhead due to instrumentation, our modification constructs an error trace by sampling the memory write operations. This is achieved by adding a knob to the Pin tool that configures the number of memory writes to skip before storing an error value. We found this useful when experimenting with time-sensitive applications such as control systems, where instrumentation overhead can spin the system out of control.

We use a visualization of the evolution of error to aid the user in identifying variables that are candidates for precision downgrade. In this visualization, each pixel represents the relative error between the original value and the shadow value of a specific memory location at a specific write operation during the run of the program. Figure 1 demonstrates an example of this visualization for the laser beam stabilizing control system mentioned earlier. The x-axis represents floating point memory locations observed during instrumentation. The y-axis represents the normalized write operations made to each memory location, and serves as a rough proxy to a time axis. In Figure 1, each memory location is normalized to be written to 800 times. Each pixel in the image represents the relative error, and a darker pixel indicates higher error.

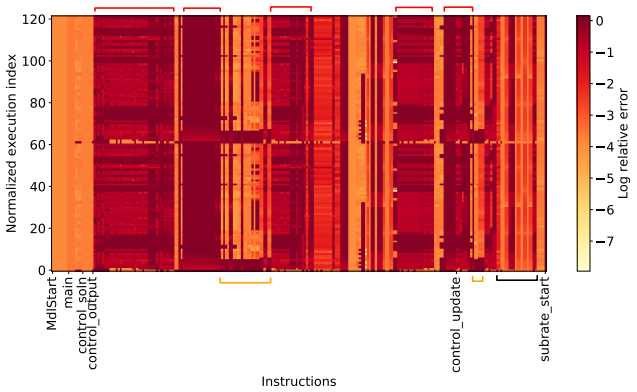

Fig. 2. Error trace per instruction. A darker pixel indicates higher error. Relative error is in $\log_{10}$ scale.

Figure 1 demonstrates multiple behaviors that should guide the precision downgrade process. For some locations, the error increases as time progresses (as we move north along the y-axis). For other locations, the error is initially high and drops shortly after the start of execution. This can be seen around memory locations 140, 280, and 600 (indicated by the red arrows at the bottom). There is also a significant portion of memory where error changes periodically (indicated by orange arrows at the top). This periodic change is indicative of repetitive behavior in the controller due to the motorized disturbance applied to the stabilizer. Thus, this visualization suggests that there are locations that can be downgraded to single precision permanently, and others that can be downgraded dynamically depending on the error trace.

2.4 Tracing Instruction Error

The next modification made to the tool is to support tracing the error caused by each floating point instruction. To do this, we record the result of every floating point instruction, tracking the output whether written to memory or an XMM register. We compare this result with the single-precision version of that instruction. Similar to memory error, we sample instruction error to avoid large overhead.

We visualize instruction error in a similar fashion to memory error. Figure 2 illustrates the error per instruction at each execution of an instruction in the laser beam stabilizer mentioned earlier. The x-axis represents instructions executed during instrumentation grouped by function. For instance, `control_output` is a function whose instructions begin under the function label and continue to the right until the `control_update` label. The y-axis represents the normalized executions of the instructions, where every instruction is normalized to be executed 120 times. The color of the pixel represents the log of the relative error for the respective instruction on the x-axis and the respective execution on the y-axis.

As shown in Figure 2, `control_output` is the largest function. The code for the laser beam stabilizer is generated using Quanser's Simulink package [19] which generates a single large function containing almost all the logic. In general, the error is mostly high (see red braces at the top of the plot), yet there are parts of the code where the error is high at the beginning but drops shortly after the start of the execution (see orange braces at the bottom of the plot). This indicates parts of the function that can be dynamically downgraded to single-precision. `control_update` is also a candidate for permanent downgrade to single-precision since it has mostly low error (black brace at the bottom of the plot).

2.5 Isolating Function Error

Simply tracing the error per instruction is not sufficient to determine the real impact of the instruction on error. This is because error propagates through instructions via their operands. We mitigate this by isolating error at the function level. Upon entering a function, we push all shadow values onto a stack, and reset the working map of memory addresses to shadow values. Any instruction within the function uses the original values in double precision, and the error we track is solely due to the rounding error of performing the instruction in single precision. The stack matches the call stack, and so calling one function from another resets the shadow values upon entering the called function, and pops the shadow values upon returning from the called function.

The process of isolating the error per function helps to make relatively coarse decisions about downgrading parts of the program to single-precision. A downgrade of a function to single-precision entails changing all stack variables to floats, all doubles passed by value to floats, and all mathematical operations to their single precision counterparts (ex. MULSD to MULSS). There are several advantages to performing the downgrade at the function level:

- The downgrade is simple enough to apply manually, which we do in the case study in Section 4. It does not require changing data types that impact other functions, which makes it easier to apply.
- The downgrade is coarse enough that it would not introduce too much overhead. Obviously this varies depending on the size of the function, but applying the downgrade at the function level will in general introduce less casting and copying of double precision variables than a downgrade at the instruction level.
- Analysis at the function level reduces the search space of interactions with respect to error. While we can measure the isolated impact of a function on error, we cannot predict the interaction when two functions are simultaneously downgraded. The error might be canceled out or magnified. Studying these interactions can result in a huge search space if done at the instruction level. We discuss further techniques to reduce the search space in the next section.
- Isolating the error at the function level helps create a set of *independent* random variables representing the error per function. Without isolation, the error of a function is dependent on other functions' errors. Such independent random variables simplify experimental design to understand interactions and compound effect on error.

3 MANAGING THE PERFORMANCE / ERROR TRADEOFF

The visualizations in the previous section illustrate how error changes as execution progresses with respect to the input data set. These illustrations will look different for different test data sets. If the user provides a data set that is representative enough of the application's real world use, results should be predictable. This is the same for any system that uses training data and is also true of performance analysis, which requires a representative workload to achieve useful optimization.

The visualizations also do not capture the tradeoff between the performance gain when downgrading a specific function, and the error resulting from the downgrade. To aid the user in managing this tradeoff, we use plots similar to the one in Figure 3.

Figure 3 illustrates the tradeoff between performance gain and average error for every function. The x-axis represents the number of executed instructions per function. This is a proxy for the performance gain achieved by downgrading the precision of each function, with a larger value indicating larger gains. The rationale for the use of this proxy is that single precision instructions are generally faster than double precision on most architectures due to fewer memory transfers, less cache occupancy, and fewer cycles for some mathematical operations. This advantage also

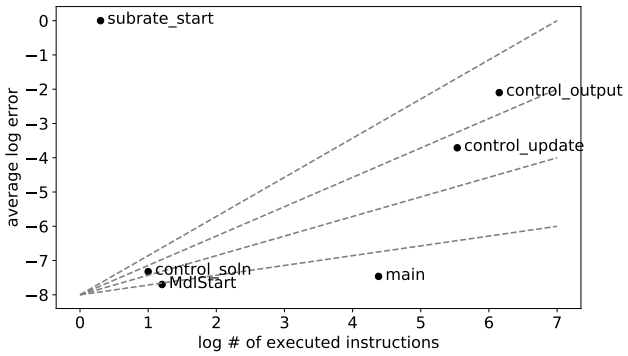

Fig. 3. Tradeoff between performance gain and error for every function. Average error is in $\log_{10}$ scale.

extends to energy, since a single precision implementation is idle for a longer time, resulting in reduced energy consumption. The y-axis represents the average error over all instructions in the function. Naturally, a lower average error is more desirable. The objective of the plot is to aid the user in identifying which functions to downgrade in precision. In general, functions residing in the lower right-hand quadrant are the best candidates for downgrading.

We use two methods to select functions for precision downgrade:

(1) **Arbitrary separation.** An arbitrary, positively-sloped line can be drawn originating from the plot's origin, under which all functions are downgraded, and above which all functions remain at double precision. An example of such lines is shown in Figure 3. The lines in the figure gradually increase in slope resulting in the inclusion of functions that have higher error but also high performance gains.

(2) **Scoring.** A score could be associated with each function in the form of

$$s = \frac{\varkappa}{\varepsilon} \tag{1}$$

where s is the score, $\varkappa$ is the number of executed instructions and ε is the average error. Functions with the highest score should be downgraded first.

An automatic search could be employed to find the optimum subset of functions to downgrade such that the compound error is minimized and the performance gain is maximized. This is somewhat similar to a knapsack where functions have value (performance gain) and weight (error), except that the combination of functions in the knapsack affects the total weight. This is similar to the work in [9, 21]. We discuss this further in Section 8.

4 APRILTAG DETECTION CASE STUDY

4.1 The Apriltag Library

Apriltags [17] are two-dimensional bar codes that are similar to QR codes and are used in a variety of applications such as robotics and augmented reality. The advantage of Apriltags over QR codes is that they encode less information, allowing more robust detection at longer distances. Apriltags can also identify a 3D position of the tag with respect to a calibrated camera. Figure 4 shows an example of an Apriltag.

Apriltag detection is implemented in a self-contained C library [2] that can process static images as well as live video. We compile it with `gcc 5.4.0` with `-O4` optimization. Our first step was to verify whether the library uses double precision floating-point arithmetic. We used Intel Pin's

`Insmix` tool to extract statistics of instruction usage. Table 1 lists the percentage of instructions executed per category. A small percentage of single precision instructions were executed but double precision dominated, accounting for almost a third of all executed instructions. This indicates a large potential for performance improvement if all or some of these double precision instructions are downgraded to single precision.

Table 1. Apriltags library instruction breakdown

Category	Percentage
General purpose	50%
MMX	1%
Single-precision SSE	5%
Double-precision SSE2	27%
Others	18%

4.2 Test Dataset

We use the PennCOSYVIO dataset [18] to test the reduced-precision versions of the Apriltags library. The dataset consists of a set of videos shot on a university campus using a GoPro camera mounted on a robot. The path of the robot is populated with physical Apriltags printed at various locations. The videos are approximately 3 minutes long at 1080p resolution.

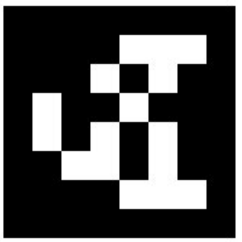

Fig. 4. An Apriltag [2].

4.3 Initial Comparison of Double vs. Single Precision

The next step was to study whether a full single precision implementation can result in acceptable error. It might be the case that converting all variables and instructions to single precision yields low error, in which case there is no need for sophisticated analysis of specific memory locations and functions. To perform this check, we converted all double precision variables in the source code to single precision and tested a set of sample images, some containing Apriltags and some not.

We observed two behaviors when testing a full single-precision version of the library. In some cases, the modified version did not terminate due to strict convergence criteria on a particular loop. Capping the iterations of this loop resulted in false negatives.

These results demonstrate that while single precision computation does often run faster than its double precision counterpart, algorithms with value-based control flow or strict termination criteria might exhibit worse performance when using single precision arithmetic. This is an important factor that should be considered when applying precision-related optimizations.

Next, we compared double and single precision implementations against a video from the dataset. The video is split into 2888 frames and we measure the number of tags detected by both implementations in each frame. False negatives are abundant and the single precision version misses 75% of all tags. False positives also occur yet are very infrequent ($< 0.1\%$).

Figure 5 demonstrates the per frame speedup gained by using single precision. The red line indicates 1x speedup, above which every point is an improvement, and below which every point is a slow down in performance. As can be seen in the plot, while there are significant improvements reaching up to 3x, the single precision implementation is not reliable. It runs longer than the

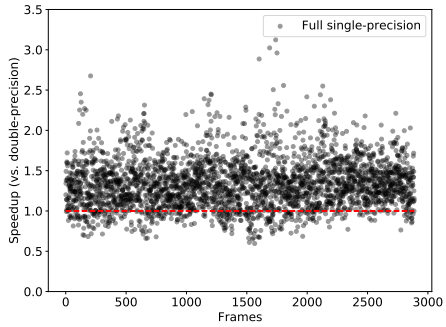

Fig. 5. Speedup of the full single-precision implementation.

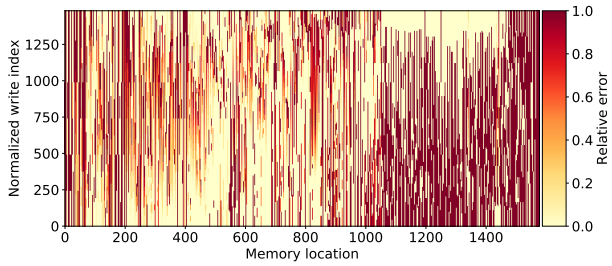

Fig. 6. Memory error trace of the Apriltags library.

original double precision implementation 11.4% of the time and generally has a higher variance. The occasional increase in run time is due to the algorithm taking more time to converge at single precision. This indicates that a trivial single-precision implementation is not appropriate because it yields high error with no reliable performance improvement.

4.4 Using the Proposed Approach to Analyze Error

The results in Section 4.3 indicate there is potential to find a sweet spot between error and performance. While a full single-precision implementation can provide attractive speedup, the accuracy drops significantly. Next, we used the analysis approach described in Section 2 to understand the behavior of the Apriltags library and find a compromise between error and performance.

4.4.1 Apriltags memory error trace. We first studied error with respect to double-precision memory locations. As explained in Section 2.3, we use our Pin tool to capture the error behavior during execution. Figure 6 illustrates the error trace per memory location. As shown in the figure, the error is generally high in all memory locations at some point during execution, making per-variable optimizations (such as the optimizations suggested in [21]) difficult.

4.4.2 Apriltags instruction error trace. We next studied the error caused by each executed instruction (see Section 2.4). Figure 7 (top) demonstrates the non isolated error per instruction grouped by function. As can be seen in the figure, there are functions that are reasonable candidates for precision downgrade, such as `fit_line`, `fit_quad`, and `quad_decode_task`.

4.4.3 Apriltags isolated instruction error trace. In Section 2.5, we introduced a method to isolate the error per function such that the measured error is not affected by propagated error passed

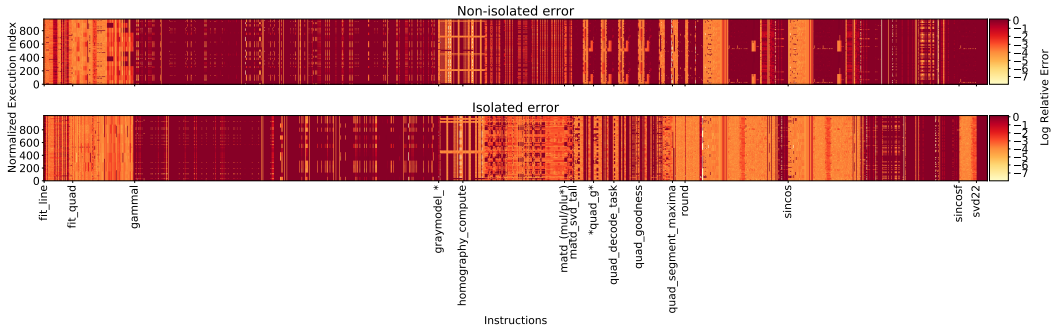

Fig. 7. Instruction error trace of the Apriltags library isolated per function and non-isolated.

to the function as parameters or accessed by the function using pointers. Upon repeating the analysis using such isolation, the error profile of some functions differed drastically. Figure 7 (bottom) illustrates the error per instruction isolated by function. As shown in the figure, functions like `quad_goodness` are now prominent candidates for downgrade due to their low error. This indicates that the error observed within `quad_goodness`'s instructions was actually a byproduct of propagated error by different functions. We can identify such functions using this simple plot, whereas tracing through the code manually is an extremely tedious process. Other functions like `homography_compute` and `quad_segment_maxima` exhibit similar behavior and are also candidates for precision downgrade.

4.4.4 Apriltags performance vs error tradeoff. Finally, we used our tool to visualize the tradeoff between performance gain and error per function. Figure 8 illustrates the tradeoff. There is a cluster of functions at the bottom right, which are strong candidates for precision downgrade due to low error and a relatively high number of executed instructions. However, while we know the isolated impact of individual functions on error, we do not know what interactions will arise when combinations of functions are downgraded simultaneously. In the next section we describe experimentation to explore this effect.

4.5 Identifying Precision Levels

Using the information provided in Figure 7 and 8, we can start downgrading parts of the program to single precision. The downgrade process is currently manual, although it could be automated as discussed in Section 8. Currently, the downgrade process includes:

- changing double-precision passed-by-value parameters to single precision, and
- changing local variables to single precision.

In this case study, each function downgrade took a few minutes and was mostly a find and replace process. The harder problem is identifying which functions to downgrade. Since there is a prohibitively large number of function combinations, we are only concerned with testing a limited number of combinations. We use the term *precision level* to refer to an implementation where a particular subset of all functions is downgraded to single precision. The following sections detail the proposed methods of defining different precision levels.

4.5.1 Levels defined by score. The first method attaches a score to a function, and converts functions with the highest score first, greedily adding more functions. We use the score formula

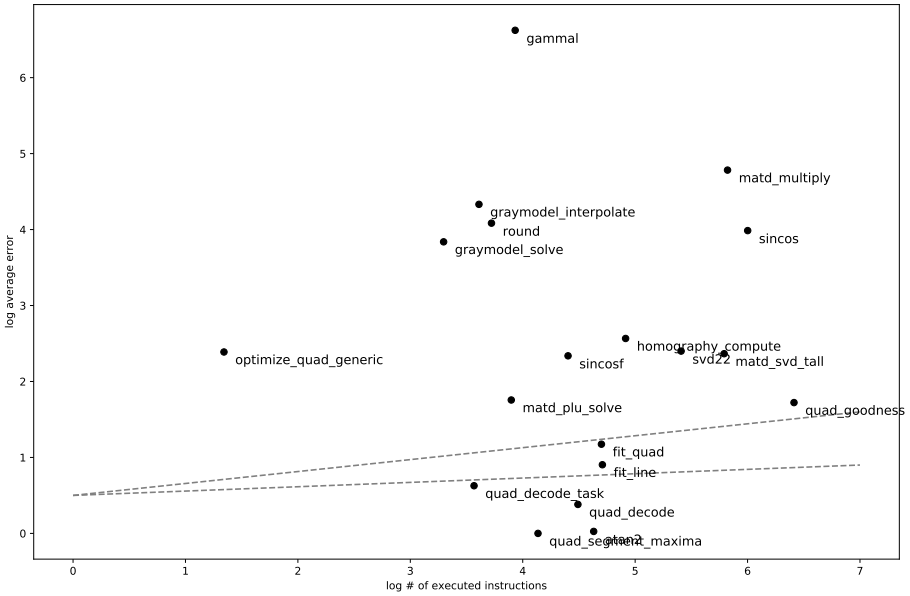

Fig. 8. Tradeoff between performance gain and error for every function in the Apriltags library.

in Equation 1 with $c = 1$. The top 5 functions are `atan2` (a `libmath` function that cannot be downgraded), `quad_goodness`, `matd_svd_tall`, `sincos` (similar to `atan2`), and `matd_multiply`.

We define two levels to experiment with levels by score:

- **Level-1**: `quad_goodness`
- **Level-2**: `quad_goodness` and `matd_svd_tall`

4.5.2 Levels defined visually. The second method is to use an arbitrary line on the tradeoff graph to decide which functions are downgraded (below the line) and which are not. An example of such lines is shown in Figure 8. This method helps make changes more coarse and can be used in combination with scores, as we will show in the next section. We use the lines in Figure 8 to define our levels visually:

- **Level-3**: `quad_decode_task`, `quad_segment_maxima`, and `quad_decode`.
- **Level-4**: `fit_line`, `fit_quad`, `quad_segment_maxima`, `quad_decode`, and `quad_decode_task`.

5 EXPERIMENTS AND RESULTS

In this section, we demonstrate how to define precision levels and ultimately find a compromise between performance and error.

5.1 Metrics

We use a set of metrics to measure the quality of an implementation from the perspective of performance as well as error. The following are the metrics we measure in each run:

- Speedup vs. the double-precision version by measuring frame processing time.
- Energy vs. the double-precision version.
- Power vs. the double-precision version.

- Number of false positives per frame (i.e., the library detects tags that do not exist or have the wrong ID).
- Number of false negatives per frame (i.e., the library fails to detect existing tags in the frame).

5.2 Experimental setting

We perform our tests on two architectures: an Intel Core i5 machine (i5-4460S CPU @ 2.90GHz, 4 cores, 6MB Cache, 8GB RAM) and a Raspberry Pi 3 (1.2 GHZ quad-core ARM Cortex A53, Broadcom VideoCore IV GPU, and 1 GB LPDDR2-900 SDRAM). Testing on Raspberry Pi illustrates the portability of the analysis, which was done on an Intel machine using Intel's Pin instrumentation tool, to a different architecture. The Raspberry Pi is widely used in robotics with some interesting applications [16]. As a test set, we used a video from [18] to verify the significance of our conclusions against thousands of frames.

5.3 Speedup results

Figure 9 shows the speedup per precision level. As can be seen, all precision levels almost always dominate the double-precision implementation. Table 2 shows the percentage of frames where single precision runs faster than double precision (*% with speedup*). The table also shows the average speedup across all frames.

There are some interesting points to observe:

- **Level-1** is the most reliable in terms of speedup, since the speedups are tightly distributed, and the number of times a frame runs slower in **Level-1** is less than 1%.
- **Level-2** improves speedup significantly. This is due to the downgrade of `matd_svd_tall` to single precision. Figure 8 indicates that this function is in the top five functions with most executed instructions. However, speedup is loosely distributed indicating that the function affects a part of the program that relies on precision to terminate. This is also supported by the function's high error (see Figure 8).
- **Level-3** and **Level-4** provide higher speedup than **Level-1** with a relatively tight distribution. However this comes at the cost of consistency.

Table 2. Apriltags levels performance and accuracy.

Level	Avg. Speedup	% with Speedup	Accuracy
Level 1	1.30x±0.09	99.30%	100%
Level 2	1.70x±0.25	98.60%	96.64%
Level 3	1.35x±0.18	98.06%	61.48%
Level 4	1.36x±0.249	95.77%	60.51%

5.4 Accuracy results

Figure 10 shows the number of false positives and negatives per level. As can be seen in the figure, **Level-1** has no false positives or negatives since tags in all frames match those in the double-precision implementation. However, the accuracy drops significantly when downgrading the **Level-3** functions as well as the **Level-4** functions. This is indicative of a compounding effect that increases error when these functions are downgraded. Table 2 lists the accuracy (# of frames with no false positives or negatives / number of frames). Thus, **Level-1** appears to achieve perfect

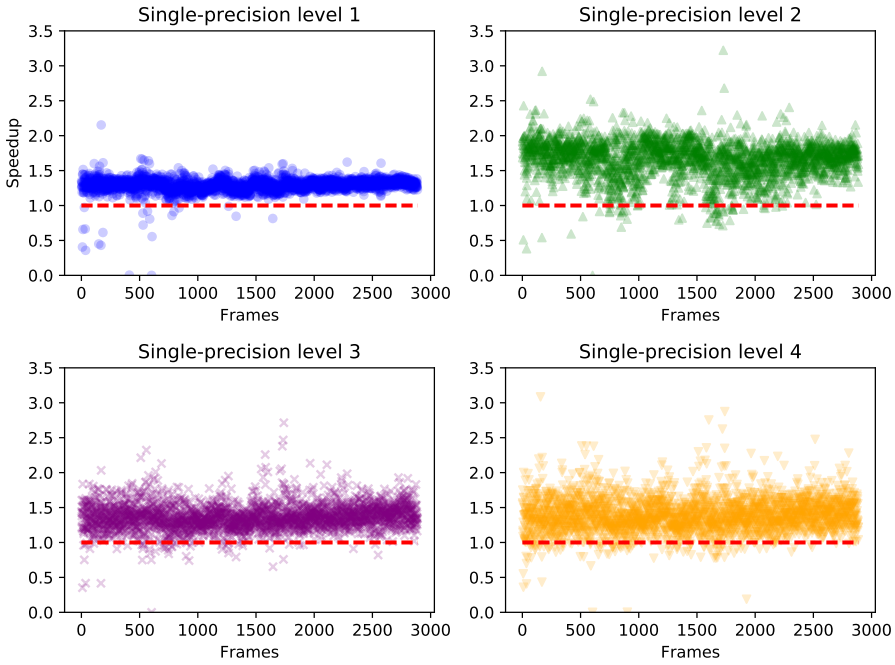

Fig. 9. Speedup per precision level.

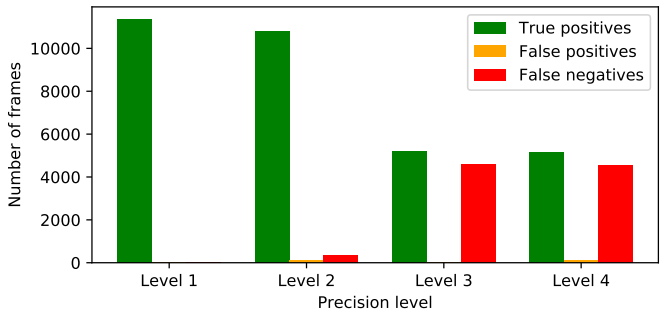

Fig. 10. False positives / negatives for each level.

accuracy (against our test dataset) while providing a 1.3x speedup. However, the decision of which level to select is to be made by the user, who can tweak their implementation to match a desired accuracy. In some cases, 96% might be acceptable, and **Level-2** could be selected for its more significant 1.7x speedup.

5.5 Energy results

Figure 11 shows the ratio between the energy consumption of each precision level versus the energy consumption of the original double precision implementation. This is measured per frame such that the comparison is fair. We use Intel's RAPL [24] to determine energy consumption. As

Table 3. Apriltags `perf` statistics.

Implementation	CPU Cycles	Instructions	IPC
Double precision	18.1×10^9	25.0×10^9	1.39
Level-1	13.9×10^9	26.7×10^9	1.92
Level-2	10.0×10^9	20.5×10^9	2.04
Level-3	13.2×10^9	25.3×10^9	1.91
Level-4	13.4×10^9	25.7×10^9	1.91

expected, performance gain also results in energy reduction. **Level-1** reduces energy consumption on average by 16%, **Level-2** by 33%, **Level-3** by 20% and **Level-4** by 21%. Similar to our remarks on speedup, **Level-1** seems to be more consistent in its improvements. This is attributed to the **Level-1** modification not affecting the iterative process of convergence, which can sometimes cause longer run times.

Figure 12 shows the ratio between the power consumption of each precision level versus that of the original double-precision implementation. We use Intel's RAPL to determine the average power consumption. Interestingly, all precision levels consume *more* power than the double-precision implementation. On average, all levels increase power consumption by around 10%. We investigated the reason behind the increase in power consumption, first exploring the CPU frequency and idle status during execution. We used `powertop` [23] to log the percentage of time the CPU spends in each frequency as well as idle. We used this log to calculate the weighted average CPU frequency. After comparing the average CPU frequency across different levels and different frames, we failed to prove that there is any significant difference between levels in terms of CPU frequency. This was confirmed using a t-test.

Next, we verified whether there is a significant difference in the number of CPU cycles used by the single precision levels versus double precision. We used `perf` to count the number of CPU cycles and instructions. Table 3 shows the average number of cycles and instructions per precision level, as well as the instructions per cycle (IPC). As shown in the table, reduced-precision levels lower the number of cycles by approximately $25\% - 55\%$ while increasing the number of instructions per cycle by $37\% - 47\%$. The increase in instructions per cycle shows more optimized pipelining due to the use of more efficient 32-bit vectorization. This explains the 10% increase in power consumption despite all implementations running at roughly the same CPU frequency.

5.6 Raspberry Pi results

We repeated our experiments on a Raspberry Pi 3 board to validate the impact of reduced precision on different architectures. The Raspberry Pi 3 is equipped with an ARM Cortex A53 (ARMv8) quad core processor. The analysis was based on Intel's instrumentation tool, running on an Intel processor, so it is interesting to see the performance and error of precision levels on an ARM processor. To measure power, we use a USB power meter. Due to timing constraints, we tested only 100 frames on the Raspberry Pi as it is significantly slower than an Intel Core i5.

Table 4 summarizes the performance of the precision levels on the Raspberry Pi. As shown in the table, speedup is similar to that on the Intel processor. Energy savings are higher on the Raspberry Pi than on the Intel processor, which is encouraging for a versatile device in embedded systems. The power consumption is generally the same across different levels versus double precision. While the Intel processor employs sophisticated pipelining technologies resulting in higher power

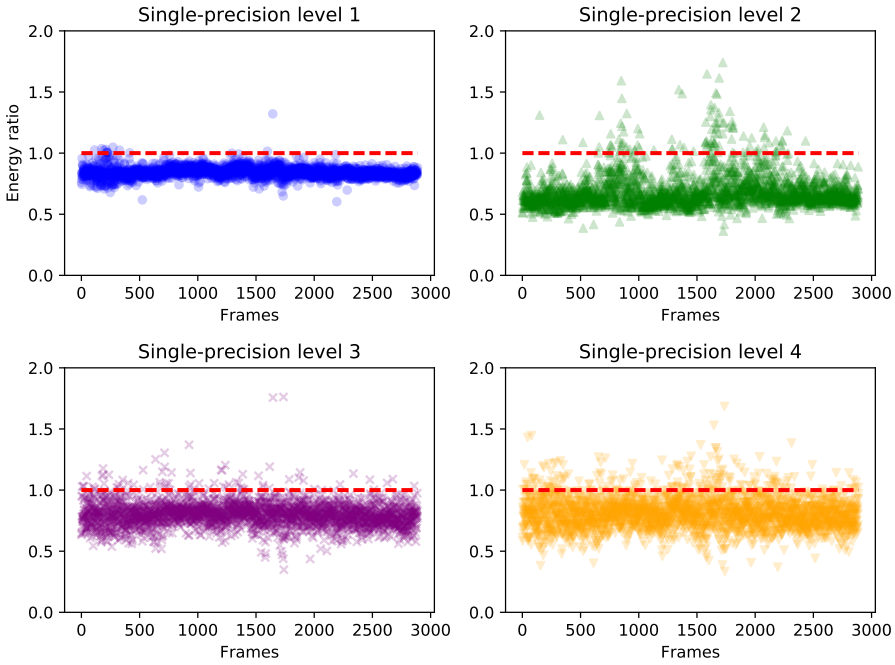

Fig. 11. Energy ratio per precision level.

Table 4. Apriltags on Raspberry Pi.

Implementation	Speedup	Energy Ratio	Power Ratio
Level-1	1.21x±0.07	0.74	0.92
Level-2	1.61x±0.23	0.59	0.91
Level-3	1.27x±0.17	0.79	1.03
Level-4	1.23x±0.25	0.78	0.97

consumption of single precision levels, the Raspberry Pi power consumption is slightly less for single precision versus double precision in most cases.

6 DISCUSSION

In this section we discuss several remarks on our proposed analysis.

Effectiveness of isolated error analysis. Since our analysis isolates error per function, inter-functional error propagation is not captured. As mentioned earlier, identifying the impact of any combination of functions on error is a combinatorial problem. Our analysis approach approximates this using isolated error, which we have shown is capable of yielding positive results. Moreover, using test data to verify different precision levels helps identify levels where actual error does not match predicted error.

Portability of analysis results. Since our tool uses Intel Pin instrumentation, it can only be used on architectures where Pin is supported. If the program cannot run on Pin-supported machines,

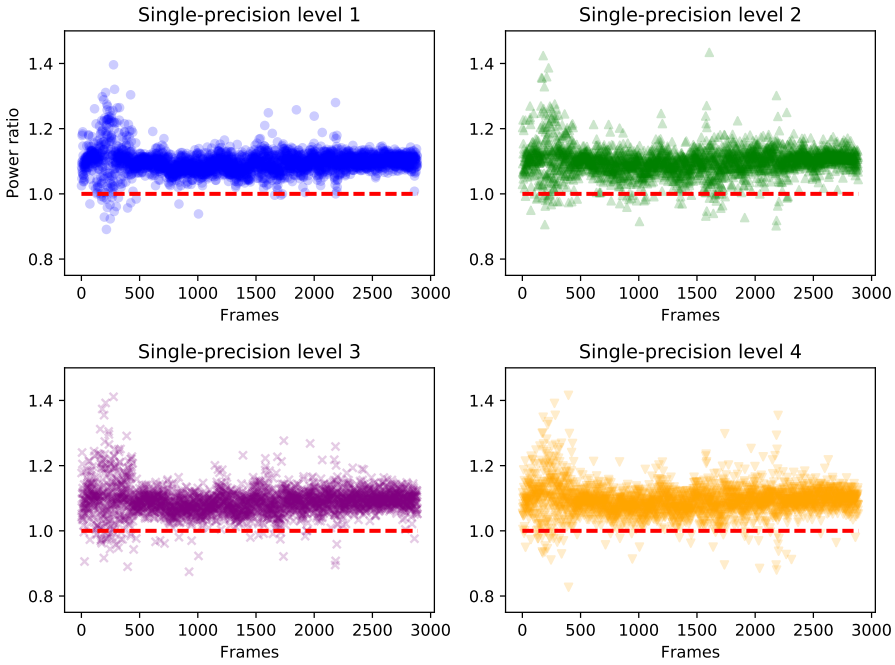

Fig. 12. Power ratio per precision level.

we will not be able to collect error data. On the other hand, the program itself can be deployed on architectures different from the one used to collect error data. If the floating-point behavior is drastically different between two architectures, our function downgrade recommendations may be erroneous. However, most architectures adhere closely to the IEEE 754 specification for floating-point arithmetic, ensuring that results are portable up to compiler-related differences. We have also demonstrated that our recommendations were useful on at least one very different architecture (the ARM-based Raspberry Pi).

Utility of methods for defining precision levels. We proposed two methods to recommend which functions to pick first when experimenting with downgrading precision: a score method and a line based separation method. The difference between the two methods is as follows: line based separation will include cheap conversions, i.e. small functions that have low impact on error and a low number of instructions. Score based selection will target the functions with the highest executions to error ratio. This can result in different outcomes depending on the application. In our case study, it appears that cheap conversions in the line based approach (levels 3 and 4) cluster into a significant drop in accuracy, due to interactions. However, the score based method results in higher accuracy and speedup. This outcome could be different for different applications, and thus it is important to use selection heuristics such as those we propose to reduce the search space and provide acceptable performance improvements.

Recursive functions. Since we reset error upon entering the function, recursion should still be supported. An interesting scenario would occur if recursion depends on the floating point value converging. We encountered a similar phenomenon but with loops. For instance, let's assume the function recurses 10 times at double precision and 20 times at single precision. Since we are using

Pin to shadow the original double precision execution, we only record error for 10 recursive calls of the function at single precision. It is safe to assume that the error at the end of these 10 calls will be higher for single precision. This higher error will proxy the cost of downgrading the precision of the recursive function, and produce an appropriate recommendation.

Effect of sampling. Sampling can result in missing key computations that affect the error reported for the function. The user is responsible for managing the tradeoff between accuracy and instrumentation time. However, since we are using average error to make recommendations, a few extreme error values will not have a strong impact on the average error. Our approach in the case study of this paper was to start at a low sampling frequency and incrementally increase the frequency until there was no change in the shape of error curve.

7 RELATED WORK

Approximate computing is an emerging approach to improve the performance and energy efficiency of embedded systems by sacrificing accuracy [6]. The work on approximate computing has targeted both hardware and software. In hardware, researchers have designed arithmetic circuitry that supports approximate computing to achieve higher energy efficiency [12]. In software, researchers have written tools to simulate program behavior on approximate computing hardware [15] and characterized the resilience of applications to approximate computing using computation models [4].

Various work has focused on optimizing precision for performance gains. A primary motivation for this work is reducing energy consumption. One approach uses qualifiers explicitly declared by developers to indicate parts of the program that can be approximately computed [22]. These qualifiers are used to guide an automated system into improving performance by managing the accuracy of the qualified parts of the program. This approach has also been explored from a runtime perspective, with dynamic monitoring and adaptation to maintain a QoS [3]. However, these techniques all require developer input. Our approach automatically provides the developer with recommendations supported by our error analysis method without first modifying the source code.

Utilizing lower-precision computation in machine learning applications is an emerging trend that is gaining traction [5]. Popular open source machine learning libraries such as Microsoft's Cognitive Toolkit [14] and Tensorflow [1] use both single and double precision. However, the selection of precision is left to the developer. In this paper, we present an automatic approach that provides more granularity than a binary whole-program precision switch.

Finally, various work has used dynamic instrumentation to optimize performance of floating point arithmetic. This includes a technique for online cancellation detection [10], an automated mixed precision search framework [9], and a rounding-error-based general precision analysis tool [8]. Others have implemented similar techniques using a compiler framework [21]. In this work, an automatic search is employed to determine the combination of variables to switch to single precision. The tool injects instructions to cast converted variables when reading from / writing to memory. Shadow value analysis has also been used to investigate the effects of reduced precision levels [11, 20]. However, these techniques either do not focus specifically on making recommendations, or do not do so from the perspective of instructions and functions. We have demonstrated that our approach provides insight into a spectrum for managing the performance/error tradeoff using simple manipulations to the source code. We have also shown in the case study that focusing only on memory is not always as effective as considering instructions.

8 CONCLUSION

In this paper, we presented a novel approach for identifying parts of a program that can tolerate lower precision with little increase in overall error. The approach focuses on the function granularity,

quantifying the error and performance gain resulting from downgrading a function to single precision. We proposed multiple approaches to select functions for reduced precision based on the quantification made by our tool.

To demonstrate the applicability of the proposed approach, we applied our methodology on a robotics vision case study. We analyzed the code of a C library responsible for detecting 2D barcodes in input video streams. We demonstrated that our approach can identify functions that have the least impact on error while providing a large performance boost. To validate our tool's analysis results, we downgraded several suggested functions to single precision and measured accuracy and performance. Results indicate that we can achieve a speedup of 1.3x at no cost to accuracy with respect to our test dataset. Results also show that a speedup of $1.7x$ can be achieved at a loss of only 4% of accuracy.

We further expanded on the utility of the proposed approach for embedded systems by studying the energy and power consumption of the reduced precision implementations. We show that we can achieve a 16% energy reduction on Intel and a 26% energy reduction on ARM with an implementation that has perfect accuracy yet reduced precision. By sacrificing 4% of accuracy, energy reduction can reach 33% on Intel, and 41% on ARM.

Our analysis of the applicability of our approach to several systems resulted in the conclusion that the proposed approach is most effective in floating-point intensive applications. Applications that utilize floating-points in computations on low-precision sensory data are not suitable candidates for our approach, since it is often the case that the sensory precision eclipses the difference between single and double floating-point precisions. Our approach is most suitable for applications involving complex mathematics, which is the case for almost all vision, speech, and AI applications.

Future work. For future work, we plan to incorporate an automated search to identify the optimal combination of functions that can be downgraded in precision. This will require applying the downgrade automatically using compiler plugins. Another direction we wish to explore is utilizing the error evolution analysis in a dynamic precision management mechanism. By identifying error trends for variables and instructions, we plan to implement a mechanism to switch precision while the program is running. Also, we wish to explore reactive precision control, in which the system calculates error online and makes precision level decisions adaptively. Finally, it should be interesting to try to apply the proposed analysis and optimizations to GPUs.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, and others. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] University of Michigan APRIL Robotics Laboratory. 2017. AprilTags. https://april.eecs.umich.edu/software/apriltag.html. (2017). [Online; accessed 03-April-2017].

[3] Woongki Baek and Trishul M Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, Vol. 45. ACM, 198–209.

[4] Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 113.

[5] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[6] Jie Han and Michael Orshansky. 2013. Approximate computing: An emerging paradigm for energy-efficient design. In *Test Symposium (ETS), 2013 18th IEEE European*. IEEE, 1–6.

[7] Ian Karlin, Abhinav Bhatele, Bradford L Chamberlain, Jonathan Cohen, Zachary Devito, Maya Gokhale, Riyaz Haque, Rich Hornung, Jeff Keasler, Dan Laney, and others. 2012. Lulesh programming model and performance ports overview. *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep* (2012).

[8] Michael O. Lam and Jeffrey K. Hollingsworth. 2016. Fine-Grained Floating-Point Precision Analysis. *International Journal of High Performance Computing Applications* (jun 2016), 1094342016652462. DOI: https://doi.org/10.1177/1094342016652462

[9] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. 2013. Automatically Adapting Programs for Mixed-Precision Floating-Point Computation. In *Proceedings of the 27th International ACM Conference on Supercomputing (ICS '13)*. ACM Press, New York, New York, USA, 369. DOI: https://doi.org/10.1145/2464996.2465018

[10] Michael O. Lam, Jeffrey K. Hollingsworth, and G.W. Stewart. 2013. Dynamic Floating-Point Cancellation Detection. *Parallel Comput.* 39, 3 (mar 2013), 146–155. DOI: https://doi.org/10.1016/j.parco.2012.08.002

[11] Michael O Lam and Barry L Rountree. 2016. Floating-point shadow value analysis. In *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools*. IEEE Press, 18–25.

[12] Cong Liu, Jie Han, and Fabrizio Lombardi. 2014. A low-power, high-performance approximate multiplier with configurable partial error recovery. In *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 95.

[13] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[14] Microsoft. 2017. Microsoft Cognitive Toolkit. https://www.microsoft.com/en-us/research/product/cognitive-toolkit/. (2017). [Online; accessed 03-April-2017].

[15] Asit K Mishra, Rajkishore Barik, and Somnath Paul. 2014. iACT: A software-hardware framework for understanding the scope of approximate computing. In *Workshop on Approximate Computing Across the System Stack (WACAS)*.

[16] Ricardo Neves and Anibal C Matos. 2013. Raspberry PI based stereo vision for small size ASVs. In *Oceans-San Diego, 2013*. IEEE, 1–6.

[17] Edwin Olson. 2011. AprilTag: A robust and flexible visual fiducial system. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 3400–3407.

[18] Bernd Pfrommer. 2017. PennCOSYVIO Data Set. https://daniilidis-group.github.io/penncosyvio/. (2017). [Online; accessed 03-April-2017].

[19] Quanser. 2017. QUARC Real-Time Control Software. http://www.quanser.com/Products/quarc. (2017). [Online; accessed 03-April-2017].

[20] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H Bailey, and David Hough. 2016. Floating-point precision tuning using blame analysis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 1074–1085.

[21] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 27.

[22] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 164–174.

[23] Intel Open Source. 2017. Powertop. https://01.org/powertop. (2017). [Online; accessed 03-April-2017].

[24] Intel Open Source. 2017. RAPL. https://01.org/rapl-power-meter. (2017). [Online; accessed 03-April-2017].