Parallel Computing 39 (2013) 146–155

Contents lists available at SciVerse ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

# Dynamic floating-point cancellation detection

Michael O. Lam *, Jeffrey K. Hollingsworth, G.W. Stewart

*Dept. of Computer Science, University of Maryland, A.V. Williams Building, College Park, MD 20742, United States*

ARTICLE INFO

*Article history:*
Available online 28 September 2012

*Keywords:*
Tools
Floating-point
Program analysis
Correctness
Debugging

ABSTRACT

As scientific computation continues to scale, it is crucial to use floating-point arithmetic processors as efficiently as possible. Lower precision allows streaming architectures to perform more operations per second and can reduce memory bandwidth pressure on all architectures. However, using a precision that is too low for a given algorithm and data set will result in inaccurate results. Thus, developers must balance speed and accuracy when choosing the floating-point precision of their subroutines and data structures. We are building tools to help developers learn about the runtime floating-point behavior of their programs, and to help them make implementation decisions regarding this behavior. We propose a tool that performs automatic binary instrumentation of floating-point code to detect mathematical cancellations. In particular, we show how our prototype can detect the variation in cancellation patterns for different pivoting strategies in Gaussian elimination, as well as how our prototype can detect a program's sensitivity to ill-conditioned input sets.



## 1. Introduction

The finite precision and roundoff error of floating-point representations have caused problems for high-performance computational scientists since the early days of computing. With the rapid rise of GPUs and other stream-based architectures, where single-precision computations are significantly faster than the corresponding double-precision computations, there is a strong motivation to reduce precision wherever possible. Likewise, single precision numbers require less storage space, which can allow more values to be retained in a local cache, reducing memory bandwidth requirements. However, many problem domains require at least double-precision arithmetic for at least part of an algorithm in order to achieve accurate and useful results.

To properly balance the competing goals of speed and accuracy, programmers must be able to estimate the sensitivity of their algorithms and data sets with respect to different precisions. There are methods for estimating the error of numerical algorithms, but they require extensive training to use correctly and often yield error bounds that are too pessimistic. The more common approach to detecting floating-point error is to re-run a program on a representative data set using a higher precision to see if the results are significantly different. This can be painful to do manually, especially if the programmer must modify the source code extensively.

We propose a framework for automatic binary instrumentation of floating-point programs with two primary goals: (1) the detection of significant digit cancellation events, and (2) execution with alternate precisions. The former uses a straightforward examination of the values involved in addition and subtraction operations. This paper focuses on the former type of analysis, and the latter forms the main thrust of our ongoing research.

* Corresponding author.
*E-mail addresses:* lam@cs.umd.edu (M.O. Lam), hollings@cs.umd.edu (J.K. Hollingsworth), stewart@cs.umd.edu (G.W. Stewart).

0167-8191/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2012.08.002

We have implemented a prototype of such a system using the DyninstAPI instrumentation toolkit. We believe our tool is useful to floating-point code developers who do not have the skills or time required to do a full manual analysis of their code. Using our tool, developers can automatically obtain a comprehensive report on the cancellations detected during their program's execution. Since our tool operates on binaries instead of source code, developers can also run the same analyses on third-party libraries without the need for source code. In this paper we present a description of our methods and preliminary examples of results obtained using our prototype.

## 2. Related work

There is a large body of work on general error analysis in the areas of numerical analysis and scientific computing. In practice, there have been several major approaches to dealing with roundoff error. The first is to simply ignore it. In many engineering applications, measurement error far exceeds any roundoff error. Thus, the standard double precision provided by current computers is usually sufficient.

The second approach is to try to quantify the error of a set of calculations *a priori* [1]. Usually this involves characterizing the error of each operation and then somehow combining them. There are two complementary approaches: *forward* and *backward* analysis. Forward error analysis begins at the input and examines how errors are magnified by each operation. *Interval arithmetic* is a variation on forward error analysis that represents each number as a worst-case range of possible values, performing all calculations on these ranges instead of individual numbers. The ranges inevitably expand as the calculations proceeded, and the range for a final answer can be quite large. Since the average-case error is rarely as bad as the worst-case, this kind of analysis is usually of little value. Backward error analysis is a separate approach that starts with the computed answer and determines the exact input that would produce it; this "fake" input can then be compared to the real input to see how different they are.

Numerical analysts have performed these types of analyses on many algorithms (see [2–5] for examples), but it is usually a difficult and tedious process. An automated solution would help considerably.

One approach to automatic error analysis is to manually insert error-tracking statements in computer code [6,7]. This augmented code calculates the error associated with the result at any given point in the calculation, maintaining it through all calculations and producing it as output along with the final result. This approach works, but is tedious and error-prone for several reasons. First, developers have to work with a numerical analyst to determine the correct error formulas. Second, if the developers ever decide to change any part of the computation, they have to ensure that they also update every corresponding error calculation. Finally, running the code without the overhead requires manually removing the tracking code.

More recently, static program analysis has provided another way to conduct error analysis [8–11]. This approach characterizes the error of mathematical operations using a set of static inference rules, allowing a compile-time analysis to determine the worst-case precision of a final result. The advantage of this approach is that it is fully automatic. Unfortunately, it is not data-sensitive; it cannot determine when an algorithm is ill-conditioned on one input set but not another. Because it is not a runtime analysis, it also cannot detect cancellation events.

FloatWatch [12] is a dynamic instrumentation approach that uses the Valgrind tool to monitor the minimum and maximum values that each memory location holds during the course of execution. While this kind of analysis reports metadata about range, it does not analyze cancellation events or allow calculations at alternate precisions.

## 3. Background

"Floating-point" is a method of representing real numbers in a finite binary format. It stores a number in a fixed-width field with three segments: (1) a sign bit, (2) an exponent field ($e$), and (3) a significand ($s$). The significand is also sometimes called the "mantissa". The actual value of the number stored is $s \cdot 2^e$. Floating-point was first used in computers in the early 1940's, and was standardized by IEEE in 1985, with the latest revision approved in 2008 [13]. The IEEE standard provides for different levels of precision by varying the field width, with the most common widths being 32 bits ("single" precision) and 64 bits ("double" precision). See Fig. 1 for a graphical representation of these formats.

Numerical cancellation occurs when an instruction subtracts two numbers that are identical in many of their digits. The identical digits are "canceled", and the resulting number has fewer significant digits than either of the operands. For example, consider the following operations:

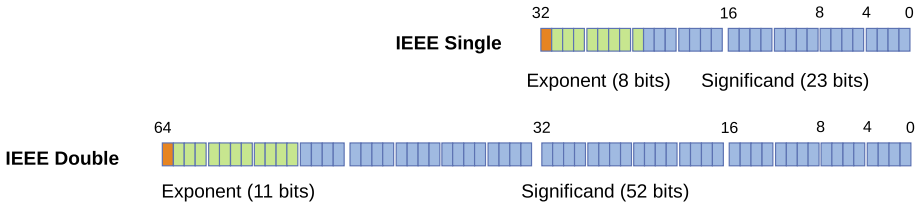


**Fig. 1.** IEEE standard formats.

```
1.613647 (7)          1.613647 (7)
−1.613635 (7)          −1.613647 (7)
0.000012 (2)          0.000000 (0)
```
(a) (b)

In the operation on the left (a), the operands all have seven significant digits, while the result only has two. In the operation on the right (b), the problem is even worse; all digits cancel and the result has no significant digits. This cancellation may seem innocuous; after all, the answer is correct! However, consider what may happen if the two numbers were not actually identical, but were rounded by previous operations. If the difference between the numbers is ever used as a scalar in a multiplication operation, for example, the result will be dramatically different (zero!) than expected.

Cancellation is something like a null pointer dereference in this respect. It can tell that there may be a problem, but not necessarily exactly where the problem is. In the case of the above examples, the actual problem might be in the routine that rounds the numbers directly before this calculation.

## 4. Methods

Our approach uses injected binary instrumentation to perform dynamic analysis of floating-point code. This analysis is automated, does not require source code, and is data-sensitive. There is of course a performance penalty, but we believe this can be mitigated in the future by optimization and tuning.

We use the DyninstAPI library [14] to insert the instrumentation. DyninstAPI can instrument in both online and offline modes. In the online mode, the tool starts the target process, pauses it, inserts instrumentation in the target's address space, and then resumes the process. In the offline mode, the tool opens the target executable, inserts instrumentation, and saves the resulting file back to disk. The resulting binary can be launched just like the original program. DyninstAPI inserts instrumentation using a trampoline-based approach, which replaces a section of executable code with a call to a *trampoline*, a newly-allocated area of code that contains the original (now relocated) instructions as well as the desired instrumentation code. Our tool augments floating-point instructions with calls to analysis routines in a dynamically linked shared library. We use the XED instruction decoder from the Intel Pin toolkit to parse floating-point instructions [15].

### 4.1. Cancellation detection

Currently, our analysis detects and reports cancellation events. To do this, we instrument every floating-point addition and subtraction operation, augmenting it with code that retrieves the operand values at runtime. Our algorithm compares the binary exponents of the operands ($exp_1$ and $exp_2$) as well as the result ($exp_r$). If the exponent of the result is smaller than the maximum of those of the two operands (i.e. $exp_r < \max(exp_1, exp_2)$), cancellation has occurred. We define the *priority* as $\max(exp_1, exp_2) - exp_r$, a measure of the severity of a cancellation. The analysis will ignore any cancellations under a given minimum threshold. Unless otherwise noted, we used a threshold of ten bits (approximately three decimal digits) for the results in this paper. If the analysis determines that the cancellation should be reported, it saves an entry to a log file. This entry contains information about the instruction, the operands, and the current execution stack. Obviously, the stack trace results will be more informative if the original executable was compiled with debug information, but this is not necessary. The analysis also maintains basic instruction execution counters for the instrumented instructions.

Since many programs produce thousands or millions of cancellations, it is impractical (and unhelpful) to report the details of every single one. Instead, we use a sample-based approach. Unfortunately, the number of cancellations that an individual instruction may produce varies wildly. In the same run, some instructions may produce fewer than ten cancellations while others produce millions. Thus, a uniform sampling strategy will not work, and we have implemented a logarithmic sampling strategy. In our tool, the first ten cancellations for each instruction are reported, then every tenth cancellation of the next thousand, then every hundred thousandth cancellation thereafter. We found that this strategy produces an amount of output that is both useful and manageable. We emphasize that all cancellations are counted and that the sampling applies only to the logging of detailed information such as operand values and stack traces.

### 4.2. Visualization

We have also created a log viewer that provides an easy-to-use interface for exploring the results of an analysis run. This viewer shows all events detected during program execution with their associated messages and stack traces. It also aggregates count and cancellation results by instruction into a single table.

The viewer also synthesizes various results to produce new statistics. Along with the raw execution and cancellation information, it also calculates the *cancellation ratio* for each instruction, which is defined as the number of cancellations divided by the number of executions. This gives an indication of how cancellation-prone a particular instruction is. The viewer also calculates the average priority (number of canceled bits) across all cancellations for each instruction. This gives an indication of how severe the cancellations induced by that instruction were.

## 5. Experiments

In this section we present several example uses of our tool to demonstrate its capabilities, usefulness, and overhead. We performed overhead experiments on a 64-bit Intel Xeon 2.4 Ghz 24-core shared memory machine with 48 GB of RAM and a local hard drive. Note that all analysis is currently single-threaded, so only one core was used at a time.

### 5.1. Simple cancellation

Our first test case is a simple example of cancellation. This sort of example is well-known to numerical analysts, and there are many workarounds. Here it serves as an introductory demonstration of our tool.

$$y = \frac{1 - \cos x}{x^2} \tag{1}$$

Fig. 2 (left side) shows the graphical representation of the function given in Eq. 1. This function is undefined at $x = 0$ since this triggers a division by zero, but as it approaches that point the function value gets infinitely close to $1/2$. In floating point, the subtraction operation in the numerator results in cancellation around $x = 0$ because $cos\ 0 = 1$. This cancellation causes the divergent behavior shown in Fig. 2 (right side). Note that the jagged appearance of the divergence is a result of the discretization of the cosine function near machine epsilon. The preferred way to avoid this behavior is to rewrite the function to avoid the cancellation. In this case, trigonometric identities allow it to be written to use the sine function, which does not suffer from the same cancellation issues at $x = 0$.

We wrote a simple program that evaluates this function at several points approaching $x = 0$ from both sides, and allowed our cancellation detector to analyze it. The tool reported all the cancellation events we expected. The output log included details about the instruction, the operands, and the number of binary digits canceled. Fig. 3 shows a screenshot of the log viewer interface. The lower portion displays all events logged during execution. Each event is displayed in the list in the lower-left corner, along with summary information about the event. Clicking on an individual event reveals additional information in the lower-right corner and also loads the source code in the top window if the debug information and the source files are available. If possible, the tool also highlights the source line containing the selected instruction. The tab selector in the middle allows access to other information, such as a view of cancellations aggregated by instruction.

This simple example confirmed our expectations and demonstrates how our tool works. The highlighted message reveals a 51-bit cancellation in the subtraction operation on line 19 of `catastrophic.c`. The two operands involved were two XMM registers with values that were both very close to 1.0 (the first was exact and the second diverged around the 16th decimal digit). Selecting the other events reveals similar details for those cancellations. Being able to examine cancellation at this level of detail is valuable in analyzing the numerical stability of a floating-point program. In this case, it alerts us that the results of the subtraction operation on line 19 may cause a cancellation of many digits. Since the resulting value is later used on the same line to scale another value, we may deduce that this code needs to be rewritten to avoid the loss of significant digits.

### 5.2. Gaussian elimination

The ability of cancellation detection to shed light on a particular algorithm has its limits. There are two principal reasons. First, almost all algorithms contain a background of trivial cancellations that can mask more significant ones. Second, some algorithms may conceal a significant cancellation under a sequence of small, harmless looking cancellations. We will now examine these limits by looking at two issues in Gaussian elimination: (1) the instability of classical Gaussian elimination without pivoting and (2) the ability of Gaussian elimination to detect ill conditioning in a positive definite matrix.

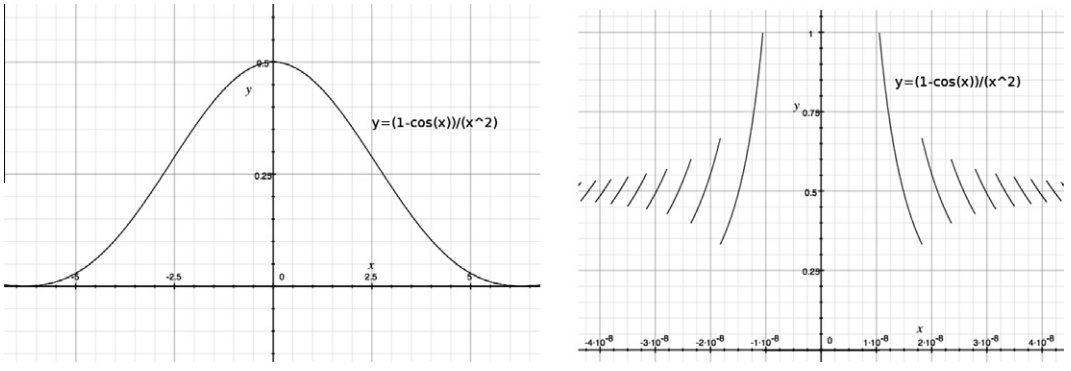


**Fig. 2.** Graphs of Eq. (1): at normal zoom (left) and zoomed to the area of interest (right).

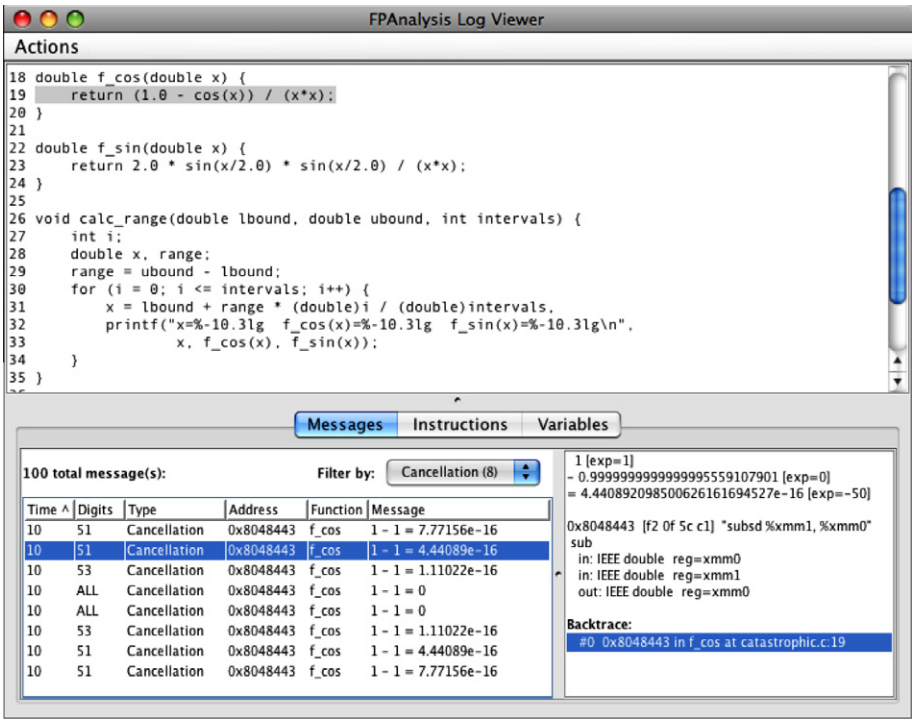


**Fig. 3.** Sample log viewer results.

```
1. perm = 1:n
2. for k=1:n
3.     [maxak, pvt] = max(abs(A(k:n,k)));
4.     A([k,pvt],:) = A([pvt,k],:);
5.     perm([k,pvt]) = perm([pvt,k]);
6.     A(k+1:n,k) = A(k+1:n,k)/A(k,k)
7.     A(k+1:n,k+1:n) = A(k+1:n,k+1:n)
                          - A(k+1:n,k)*A(k,k+1:n);
8. end
```

**Fig. 4.** Classical Gaussian elimination with partial pivoting.

Matlab code for Gaussian elimination with partial pivoting is given in Fig. 4. The result of this code is a unit lower triangular matrix.

$L = \texttt{tril}(A,-1) + \texttt{diag}(\texttt{ones}(1,n))$

and an upper triangular matrix $U = \texttt{triu}(A)$ such that

$A(\texttt{perm},:) = L * U.$

The purpose of the partial pivoting in lines 3–5 of Fig. 4 is nominally to avoid division by zero in line 6. However, if $A(k,k)$ is small, the algorithm will produce inaccurate results and cancellation will signal this situation. To see why, consider what happens when we omit lines 3–5 in Fig. 4 and apply it to the matrix shown in Fig. 6(a).[1] Note that after line 6 the elements of $A(2{:}4,1)/A(1,1)$ are all $10^3$, so that we can expect a large matrix when we compute the Schur complement $A(2{:}4,2{:}4)$. Indeed, we get the matrix shown in Fig. 6(b). Since all the numbers in the Schur complement are approximately

[1] The computations for this example were done in six-digit decimal floating-point arithmetic using the Matlab package Flap [16].

```
1.  for k = 2:n
2.      A(k,1:k-1) = A(k,1:k-1)/triu(A(1:k-1,1:k-1));
3.      A(1:k-1,k) = (tril(A(1:k-1,1:k-1),-1)
                        + diag(ones(1,k-1)))*A(1:k-1,k);
4.      dot = A(k,1:k-1)*A(1:k-1,k);
5.      A(k,k) = A(k,k) - dot;
6.  end
```

**Fig. 5.** Bordered algorithm for Gaussian elimination.

$$\begin{bmatrix} 1.00000\cdot 10^{-03} & 1.00000\cdot 10^{+00} & 1.00000\cdot 10^{+00} & 1.00000\cdot 10^{+00} \\ 1.00000\cdot 10^{+00} & -7.92207\cdot 10^{-01} & -3.57117\cdot 10^{-02} & -6.78735\cdot 10^{-01} \\ 1.00000\cdot 10^{+00} & -9.59492\cdot 10^{-01} & -8.49129\cdot 10^{-01} & -7.57740\cdot 10^{-01} \\ 1.00000\cdot 10^{+00} & -6.55741\cdot 10^{-01} & -9.33993\cdot 10^{-01} & -7.43132\cdot 10^{-01} \end{bmatrix}$$

(a)

$$\begin{bmatrix} -1.00079\cdot 10^{+03} & -1.00004\cdot 10^{+03} & -1.00068\cdot 10^{+03} \\ -1.00096\cdot 10^{+03} & -1.00085\cdot 10^{+03} & -1.00076\cdot 10^{+03} \\ -1.00066\cdot 10^{+03} & -1.00093\cdot 10^{+03} & -1.00074\cdot 10^{+03} \end{bmatrix}$$

(b)

$$\begin{bmatrix} -6.40000\cdot 10^{-01} & 9.00000\cdot 10^{-02} \\ -1.02000\cdot 10^{+00} & -1.90000\cdot 10^{-01} \end{bmatrix}$$

(c)

**Fig. 6.** Example of cancellation in Gaussian elimination.

$-10^3$, we can expect cancellation when we compute the next Schur complement, as shown in Fig. 6(c). The numbers in this matrix are back to the original magnitude, but as the trailing zeros indicate, they now have at most two digits of accuracy.

It is worth noting that the cancellation itself introduces no significant errors. The damage was done in passing from the data shown in Fig. 6(a) to that of Fig. 6(b). The subtraction of $10^3$ from the elements of `A(2:4,2:4)` caused about four digits to be lost in each of the elements. It is important to emphasize that cancellation is usually not a killer but instead a death certificate that reveals an earlier loss of information. As mentioned earlier, cancellation is a lot like a null pointer dereference, where the null pointer exception is not the problem, but rather the notification of an earlier error.

To see how well cancellation due to lack of pivoting was detected by our system, we performed the following experiment. A matrix $A$ of order $n$ was generated that had a pivot of size $10^{-s}$ at stage $p$ of the elimination. (In the example above, $n = 4,\ s = 3$, and $p = 1$.) We then ran the elimination and counted cancellations. We set the threshold (the number of bits required for a cancellation to register) at $\log_2 10^{s-2}$ rounded to the nearest integer greater than zero. This means that we regard cancellations of greater than $s-2$ decimal digits as significant. As the threshold is increased over this value we increasingly risk missing cancellations due to the bad pivot. As it is decreased we increase the risk of including cancellations not due to the pivot (i.e. background cancellations).

We can compute the number of expected cancellations due to the bad pivot by determining the dimensions of the array in which the cancellation will occur. It is of order $n-p-2$, and hence the expected number of cancellations is $(n-p-2)^2$.

We can also estimate the background cancellation. The matrix $A$ was generated in such a way as to damp cancellation before $k = p$. If we then stop the process after the cancellation (at $k = p+1$) and if $p$ is not large, the cancellation count will be a good estimate of the cancellation due to the bad pivot.

The results are summarized in Fig. 7. The rows labeled "Count" give the cancellation counts for the entire elimination while the rows labeled "Trunc" give the count for the truncated elimination. The rows labeled "Est" contain the cancellation count estimated by the formula $(n-p-2)^2$.

In the first column, the counts considerably overestimate the amount of cancellation due to the bad pivot. This is because of the small value of the threshold. In the remaining three columns, all counts are in reasonable agreement. This suggests that if care is taken to keep the threshold high enough, one can detect the effects of a reasonably small pivot. A potential application for this method is elimination on sparse matrices, where the ability to pivot is circumscribed.

Our second example concerns the ability of Gaussian elimination to detect ill-conditioning. To avoid the complications of pivoting, we worked with positive definite matrices, for which pivoting is not required to guarantee stability.

| log(size) | −2 | −4 | −6 | −8 |
|---|---|---|---|---|
| Threshold | 1 | 7 | 13 | 17 |
| $n = 10$ | | | | |
| Count | 66 | 37 | 37 | 34 |
| Trunc | 55 | 37 | 37 | 34 |
| Est | 25 | 25 | 25 | 25 |
| $n = 15$ | | | | |
| Count | 225 | 123 | 122 | 122 |
| Trunc | 154 | 122 | 122 | 122 |
| Est | 100 | 100 | 100 | 100 |
| $n = 20$ | | | | |
| Count | 663 | 247 | 252 | 257 |
| Trunc | 298 | 245 | 252 | 257 |
| Est | 225 | 225 | 225 | 225 |
| $n = 25$ | | | | |
| Count | 1227 | 394 | 423 | 441 |
| Trunc | 447 | 381 | 423 | 441 |
| Est | 400 | 400 | 400 | 400 |

**Fig. 7.** Cancellation for unpivoted Gaussian elimination.

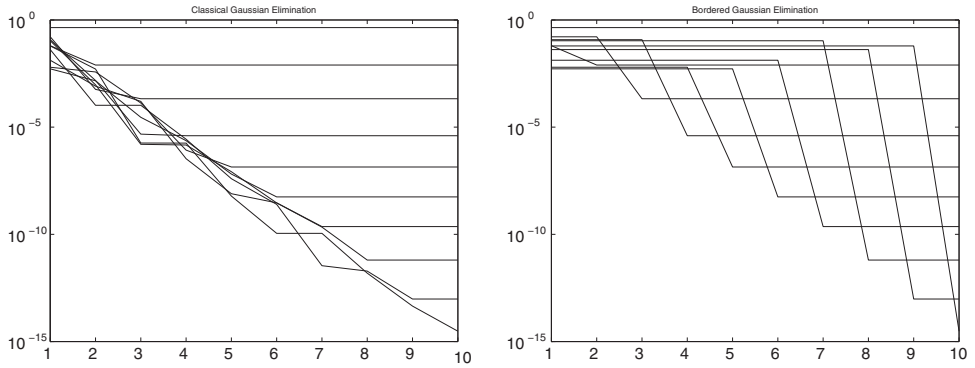


**Fig. 8.** Diagonal elements for classical (left) and bordered (right) Gaussian elimination.

Let us suppose that we have a positive definite matrix $A$ whose eigenvalues descend in geometric progression from one to $10^{-\text{logkap}}$, where logkap is a constant that we varied in our experiments. Then $A$ has the condition number $\kappa = \|A\|\|A^{-1}\| = 10^{\text{logkap}}$. If Gaussian elimination is used to compute the LU-factorization of $A$, then the diagonals of $U$ will more or less track the eigenvalues of $A$.[2] Since the elements of $A$ are of order one, the diagonals of $U$, which become progressively smaller, are calculated with cancellation. The problem is how well these cancellations will be detected.

A difficulty here is that Gaussian elimination has many variants. Consider, for example, the code in Fig. 5. It performs Gaussian elimination by bordering; after step `k`, `A (1:k,1:k)` contains the LU factorization of the original submatrix `A (1:k,1:k)`. Numerically the algorithms are almost identical, even to the effects of rounding error. However, they exhibit cancellation in different ways. The plots in Fig. 8 contain histories of the diagonal elements of the reduction of the matrix $A$ described above with $n = 10$ and logkap $= 15$. The x-axis is the step in the elimination and the y-axis is the value of the diagonal element in question.

The difference in the behaviors of the two methods is remarkable. For classical Gaussian elimination the first diagonal remains constant during the first iteration while the others decrease by roughly the same amount. In the second iteration, the second diagonal peels off and remains constant, while the others decrease. Thus, in the $i$th iteration, the $i$th diagonal becomes constant while all lower diagonals continue to decrease. In the end, each diagonal contains a rough approximation to its corresponding eigenvalue. In the border variant, on the other hand, all the diagonals remain constant during the $i$th iteration, except the $i$th value which drops to its final value and remains constant thereafter. Thus each diagonal makes only one transition (from its initial value to its final value). Naturally, the initial and final values for both methods are identical. To summarize, the classical method has many small cancellations while the bordered method has fewer and larger cancellations even though they end up at the same values.

[2] Because Gaussian elimination is not one of the best ways to estimate condition, we have preprocessed $A$ so that the tracking is improved.

| threshold | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|
| logkap | C | B | C | B | C | B | C | B | C | B |
| 5 | 14 | 8 | 8 | 7 | 1 | 6 | 0 | 5 | 0 | 4 |
| 10 | 29 | 8 | 23 | 8 | 16 | 7 | 11 | 7 | 3 | 6 |
| 15 | 39 | 9 | 33 | 9 | 27 | 9 | 21 | 8 | 17 | 8 |

**Fig. 9.** Cancellation counts for classical (C) and bordered (B) Gaussian elimination.

| Name | Original | Overhead |
|---|---|---|
| soplex | 1s | 10X |
| povray | 2s | 85X |
| lbm | 20s | 70X |
| milc | 44s | 75X |
| namd | 95s | 160X |

**Fig. 10.** Analysis overheads on selected SPEC benchmarks for instruction count and cancellation detection.

All this suggests that cancellation detection will work better for the bordered variant. Fig. 9 shows cancellation counts for both versions for various values of the cancellation detection threshold and logkap. It is easy to see by counting the drops in the graph for the border method that it should register nine cancellations, which it does unless the threshold is too high or logkap is too small. Ideally, classical Gaussian elimination should register 45 counts: nine in the first step, eight in the second, seven in the third, etc. However, a look at the plot shows that the sizes of the cancellations varies irregularly, so that there are small ones that may fall by the wayside due to being under our priority threshold. Only with logkap equal to 15 and a threshold of one bit, does it come near 45.

What is to be learned from these experiments? First, that it is important to vary the threshold. Most computations have a background of small cancellations, which will overwhelm more important cancellations if the threshold is set too low. Trying different thresholds may give a better view of what is happening. Second, and corollary to the first, cancellations near the background cannot be made to stand out. In particular if a large cancellation is obtained by a sequence of smaller cancellations, it may go undetected. Our classical Gaussian elimination experiment is an example. Third, cancellation detection is not a panacea. It requires interpretation by someone who is familiar with the algorithm in question. Nonetheless, the experiments also suggest that cancellation detection, properly employed, can find trouble spots in an algorithm or program.

Finally, we note that not all cancellations are bad. A good example is the computation of a residual to determine the convergence of an iterative method. Since a small residual means convergence, any cancellation in computing it means something has gone right.

### 5.3. Approximate nearest neighbor

To investigate the ability of our tool to detect change in the cancellation behavior of a program based on input data, we examined an approximate nearest-neighbor software library called ANN [17]. This computational geometry library takes as inputs (1) a series of data points and (2) a series of query points. The software then finds the nearest data point neighbor (by Euclidean distance) to each query point using an approximate algorithm. This program is of interest to researchers in high-performance computing (HPC) as well as computational geometry. Algorithms like ANN are often used in HPC for auto-tuning, image processing (classification and pattern recognition), and DNA sequencing.

We ran this program instrumented with our cancellation analysis twice with different sets of points. Each set included 500,000 data points and 5000 query points. The first data set was composed of points randomly generated uniformly throughout the square defined by x- and y-coordinate ranges of $[-1, 1]$. The second data set was composed of points randomly generated very close to the same square (i.e. most x- and y-coordinates were nearly identical, and close to either $-1$ or 1). The expectation was that the second input would lead to many more cancellations for certain instructions in the distance calculation, since the coordinates are much closer.

This expectation was confirmed. The first data set caused cancellation in less than 1% of the executions of the instructions of interest, and the average number of canceled bits was less than 15. The second data set caused cancellations in 100% of the executions for the same instructions, and the average number of canceled bits was 46. This shows that the tool can expose differences in floating-point behavior on the same code resulting from varying data sets, something that static analysis techniques cannot do.

### 5.4. SPEC benchmarks

To demonstrate our tool's ability to handle larger programs, we also ran it on the SPEC CPU2006 benchmark suite [18]. We then ran our cancellation detection analysis using the provided "test" data sets. We used these smaller sets so that we could complete the analyses in a reasonable amount of time. We expect that the results for the larger data sets will be comparable.

The instrumented benchmarks experienced a 10–160X overhead, which is large but not impractical for occasional analysis. Fig. 10 shows specific overheads on selected benchmarks.

The most common result was that most cancellations occurred in only a few of the floating-point instructions: usually fewer than twenty instructions out of hundreds. Often, there were several instructions that caused cancellations 100% of the time. Without domain-specific knowledge, it is difficult to know whether these cancellations indicate a larger problem in the code.

Another interesting discovery was a section in the "povray" (ray-tracer) benchmark where there is cancellation in a color calculation. In this routine, given values were subtracted from 1.0 to give percentage components in red, green, and blue. Thus, complete cancellation in all three variables indicates the color black.

## 6. Discussion

Our approach has several advantages. It is automatic, making it easy for programmers to evaluate their software as they develop and test it. Since our analysis operates on compiled binaries rather than source code or an intermediate representation, we include all effects resulting from compiler optimizations, and we can provide results for closed-source shared libraries. In addition, the tool provides *data-sensitive* results, meaning that our tool can help reveal data sets for which a particular algorithm is ill-conditioned.

The significant disadvantage of our approach is the added overhead. We believe that this overhead can be reduced by streamlining our instrumentation and by performing data flow analysis to reduce the number of instructions that need to be instrumented. Another potential disadvantage is that our tool requires a data set to produce results; this disadvantage is inherent to any dynamic, runtime-based approach.

## 7. Future work

There are several opportunities for improving the performance overhead of our analysis. Currently, Dyninst does not directly support inserting floating-point code, so we insert calls to library functions to do the floating-point operations. This incurs the high overhead of a function call and could be avoided if floating-point support were directly available in Dyninst. In addition, the logging code is largely unoptimized. Each cancellation is logged independently, and the reports contain much duplicate information. This process could be optimized by batching log events and reducing the storage of duplicate information.

Moving forward, we are currently working on implementing the alternate-precision analysis portion of our tool, which will allow software designers to automatically reconfigure their programs with mixed precision. Our approach involves replacing floating-point values and instructions in a program in order to simulate converting a portion of a program from double precision to single precision. This allows the developer to experiment with their program's sensitivity to the precision level used. Ultimately, our goal is to develop techniques and tools that implement a feedback loop for automatically tuning the precision level of a target application. For instance, a program might only require double precision for a few accumulation or residual calculations in order to maintain a desired level of accuracy; in this case, the rest of the program may be run in single precision for the increased speed and reduced memory requirements.

Finally, we are also investigating techniques for extending this work to multiprocessing contexts with multiple threads, cores, and nodes. To be useful in these contexts, the analysis must be able to scale and aggregate results obtained from all processing units. This may require novel reporting and visualization methods.

## 8. Conclusion

We have developed a runtime cancellation detector and demonstrated that it works on small, medium, and large examples. It is automatic and provides data-sensitive cancellation results. We believe it is already a useful tool for code developers. We envision this tool as the first component of a complete suite of tools for dynamically analyzing floating-point rounding error and for isolating problems detected.

## Acknowledgments

This paper is an expanded version of a workshop paper for the Workshop on High-performance Infrastructure for Scalable Tools (WHIST) 2011.

This work supported in part by DOE Grants DE-CFC02-01ER25489, DE-FG02-01ER25510 and DE-FC02-06ER25763.

## References

[1] N.J. Higham, Accuracy and Stability of Numerical Algorithms, second ed., SIAM Philadelphia, 2002.
[2] J.H. Wilkinson, Rounding Errors in Algebraic Processes, Prentice-Hall, Inc., 1964.
[3] T. Kaneko, B. Liu, On local roundoff errors in floating-point arithmetic, J. ACM 20 (1973) 391–398.
[4] T.I. Laakso, L.B. Jackson, Bounds for floating-point roundoff noise, IEEE Trans. circuits syst. 41 (1994) 424–426.

[5] W. Kraemer, A priori worst case error bounds for floating-point computations, IEEE Trans. Comput. 47 (1998) 750–756.
[6] J.H. Wilkinson, Error analysis revisited, IMA Bull. 22 (1986) 192–200.
[7] W. Kahan, Pracniques: further remarks on reducing truncation errors, Commun. ACM 8 (1965) 40.
[8] M. Martel, Propagation of roundoff errors in finite precision computations: a semantics approach, Program. Languages Syst. (2002) 159–186.
[9] S.P. Eric Goubault, Matthieu Martel, Asserting the precision of floating-point computations: A simple abstract interpreter, Program. Languages Syst. (2002) 287–306.
[10] M. Martel, Semantics-based transformation of arithmetic expressions, Stat. Anal. (2007) 298–314.
[11] M. Martel, Program transformation for numerical precision, in: PEPM '09: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, ACM, New York, NY, USA, 2009, pp. 101–110.
[12] A. Brown, P. Kelly, W. Luk, Profiling floating point value ranges for reconfigurable implementation, in: Workshop on Reconfigurable Computing, HiPEAC, 2007.
[13] I.T. P754, ANSI/IEEE 754–1985, Standard for Binary Floating-Point Arithmetic, IEEE, New York, 1985.
[14] B. Buck, J.K. Hollingsworth, An api for runtime code patching, Int. J. High Perform. Comput. Appl. 14 (2000) 317–329.
[15] XED, X86 encoder decoder, Published 12 October 2011. Accessed 23 August 2012. <http://software.intel.com/sites/landingpage/pintool/docs/53271/xed/html/>.
[16] FLAP, Flap: a matlab package for adjustable precision floating-point arithmetic, Published 30 January 2009. Accessed 23 August 2012. <http://www.cs.umd.edu/~stewart/flap/flap.html>.
[17] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, A. Wu, An optimal algorithm for approximate nearest neighbor searching, J. ACM 45 (1998) 891–923, http://dx.doi.org/10.1145/293347.293348.
[18] Spec, CPU 2006, Spec cpu2006 benchmark, Published 24 August 2006. Accessed 23 August 2012. <http://www.spec.org/cpu2006/>.