Dynamic Floating-Point Cancellation Detection

Michael O. Lam
Department of Computer Science
University of Maryland, College Park
Email: lam@cs.umd.edu

April 20, 2010

Abstract

Floating-point rounding error is a well-known problem in numerical computation that distorts results and is difficult to analyze accurately. We propose a tool that performs automatic binary instrumentation of floating-point code to detect cancellations and to run side-by-side calculations in alternate precisions. The results of this analysis can help developers find areas of their code that are causing a loss of precision. In the future, it will also point out where reduced precision could be used to achieve a faster running time without losing accuracy. In this paper we explain the techniques and present several results, focusing on cancellation detection.

1 Introduction

The finite precision and roundoff error of floating-point representations have caused headaches for computational scientists since the early days of computing. 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. With GPUs and other stream-based architectures, where single-precision computations are significantly faster than the corresponding double-precision computations, there is strong motivation to reduce precision wherever possible. Likewise (and on all computers), single precision numbers require less space. Saving space can allow more values to be stored in a given cache size and reduce memory bandwidth requirements.

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. The latter uses *shadow-value analysis*, a technique that performs side-by-side computations for every floating-point instruction in the original program. 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, they 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. Our initial work focuses primarily on the detection of cancellations.

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, the 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*. 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 [15, 8, 10, 9] 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 [16, 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 [12, 6, 13, 14]. 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 suffers from the same problem as forward error analysis; it is not data-sensitive, which means that 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 [4] 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 do full shadow value calculations at alternate precisions.

3 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. Currently, we have implemented a cancellation detector, and we are working on a shadow value analysis engine that will allow developers to automatically run their programs in an alterate precison.

We use the DyninstAPI library [5] to insert the instrumentation. DyninstAPI supports doing this in both online and offline modes. In the online mode, the tool starts the target process, pauses it, inserts instrumentation, 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 run identically to 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 which 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 [11, 2].

3.1 Cancellation Detection

Currently, our main type of 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, there is a large discrepancy between the number of cancellations at various instructions. In the same run, some instructions may produce fewer than ten cancellations while others produce millions. Thus, a uniform sampling strategy will not work. 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.

3.2 Visualization

We have also created a log viewer that provides a 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.

4 Experiments

In this section we present several example uses of our tool to demonstrate its capabilities, usefulness, and overhead. We did all of the experiments on a single 32-bit Linux workstation with quad-core Intel x86 processors, 4GB RAM, and a network-mounted hard drive.

4.1 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.

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

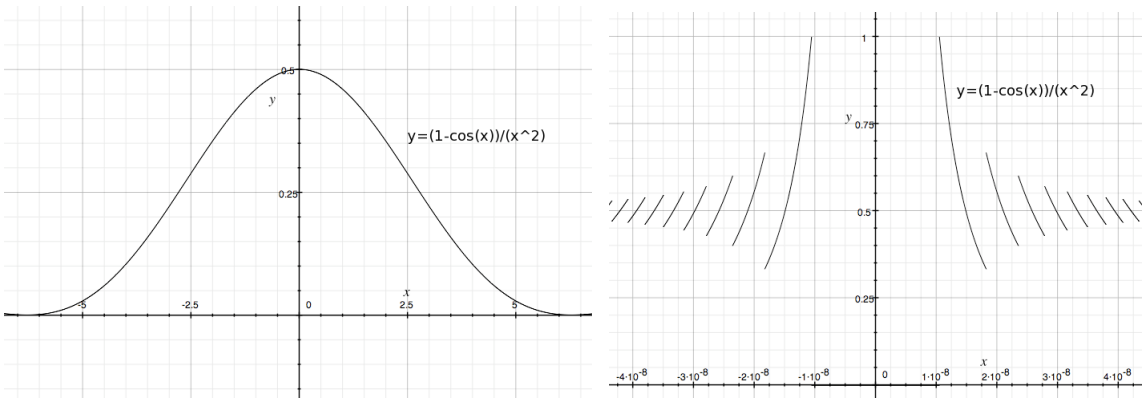

Figure 1: Graphs of Equation 1: 1a at normal zoom and 1b zoomed to the area of interest.

Fig. 1a shows the graphical representation of the function given in Equation 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. 1b. 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 ran our cancellation detector on 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. 2 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, and a list of shadow value analysis results.

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 sixteenth 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 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.

4.2 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 [3]. 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

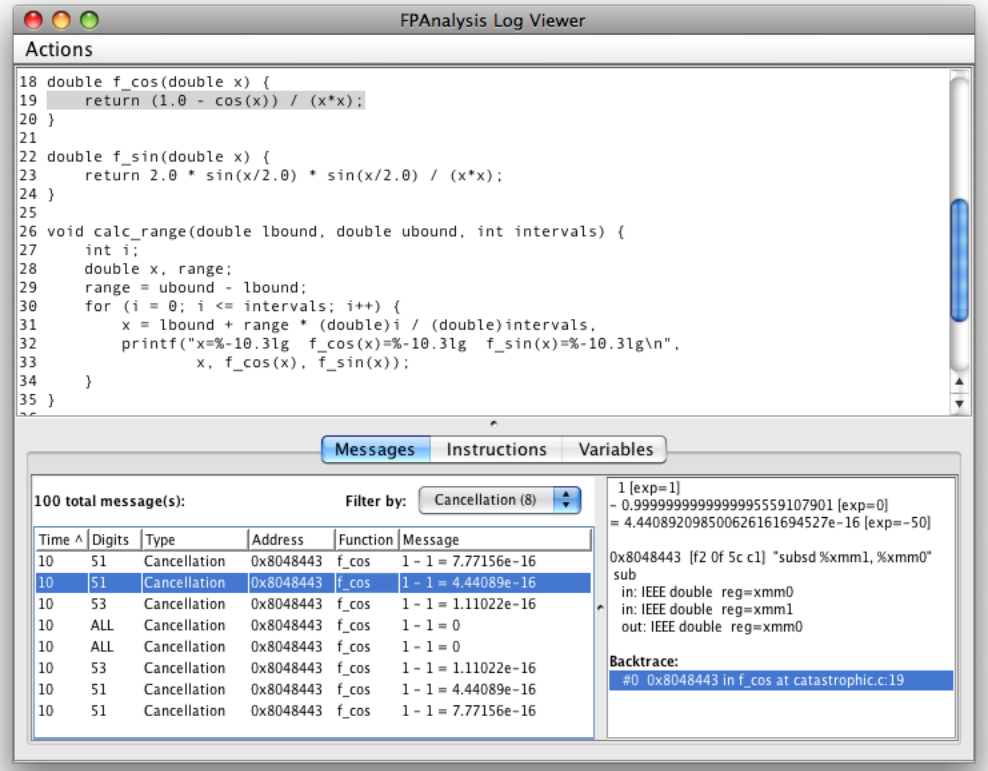

Figure 2: Sample log viewer results

Name	Count	Cancel
lbm	170X	290X
milc	220X	310X
namd	280X	420X
povray	120X	180X
soplex	20X	20X

Figure 3: Analysis overheads on selected SPEC benchmarks for instruction count ("Count") and cancellation detection ("Cancel").

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 5,000 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 error on the same code resulting from varying data sets, something that static analysis techniques cannot do.

4.3 SPEC Benchmarks

To demonstrate our tool's ability to handle larger programs, we also ran it on the SPEC CPU2006 benchmark suite [1]. 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 100-500X overhead, which is large but not impractical. Fig. 3 shows specific overheads on selected benchmarks.

The most common result was that most cancellations occurred in a few of the floating-point instructions: usually fewer than twenty instructions. 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. We are currently investigating whether these cancellations are significant.

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.

5 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. Finally, since DyninstAPI supports a wide variety of platforms, our tool works on any platform that DyninstAPI

supports. This currently includes Linux on x86/AMD64, Itanium, or PowerPC (32- and 64-bit), as well as Sparc on Solaris, Windows on x86, and AIX on PowerPC (32- and 64-bit).

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 disadvantage is that our tool requires a data set to produce results; this disadvantage is inherent to our runtime-based approach.

6 Future Work

A short term area of future work is to study the appropriate threshold value for our priority parameter. Based on our study of Gaussian elimination, it appears that a tool that automatically runs a program several times with different threshold values would be useful.

A long term area of future work is to add shadow value analysis, which will permit alternate-precision floating-point instructions to execute alongside the original program. Shadow value analysis creates a mapping between registers and memory locations that hold floating-point values and corresponding shadow value entries. These shadow values can contain alternate precisions and are updated every time a floating-point operation occurs. For example, if two floating-point numbers are added, then the corresponding shadow values are added (in the alternate precision). After the program finishes, the analysis outputs all or part of the shadow value table, reporting both shadow and actual values as well as the difference between them.

Shadow value analysis requires substantially more complex instrumentation than cancellation detection. First, it requires specialized handling of nearly all floating-point instructions, rather than a small subset as with cancellation detection. Second, the analysis must follow values through all data movement operations, even when the movement is performed by non-floating-point instructions (integer moves, `memcpy`, etc.). We plan to add support to our instrumentation system soon to provide these features.

7 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.

Acknowledgements

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

References

[1] Spec cpu2006 benchmark. http://www.spec.org/cpu2006/. accessed 23 february 2010.

[2] X86 encoder decoder. http://rogue.colorado.edu/pin/docs/20751/ xed/html/main.html. accessed 5 december 2008.

[3] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *J. ACM*, 45:891–923, 1998. (doi: 10.1145/293347.293348).

[4] A. Brown, P. Kelly, and W. Luk. Profiling floating point value ranges for reconfigurable implementation. 2007.

[5] B. Buck and J. K. Hollingsworth. An api for runtime code patching. *The International Journal of High Performance Computing Applications*, 14:317–329, 2000.

[6] S. P. Eric Goubault, Matthieu Martel. Asserting the precision of floating-point computations: A simple abstract interpreter. *Programming Languages and Systems*, pages 287–306, 2002.

[7] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, 1965.

[8] T. Kaneko and B. Liu. On local roundoff errors in floating-point arithmetic. *J. ACM*, 20(3):391–398, 1973.

[9] W. Kraemer. A priori worst case error bounds for floating-point computations. *IEEE transactions on computers*, 47(7):750–756.

[10] T. I. Laakso and L. B. Jackson. Bounds for floating-point roundoff noise. *IEEE transactions on circuits and systems*, 41(6):424–426, 1994.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[12] M. Martel. Propagation of roundoff errors in finite precision computations: A semantics approach. *Programming Languages and Systems*, pages 159–186, 2002.

[13] M. Martel. Semantics-based transformation of arithmetic expressions. *Static Analysis*, pages 298–314, 2007.

[14] M. Martel. Program transformation for numerical precision. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 101–110, New York, NY, USA, 2009. ACM.

[15] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Inc., 1964.

[16] J. H. Wilkinson. Error analysis revisited. *IMA Bulletin*, 22(11/12):192–200, 1986.