Project Final Report:

Correctness Tools for Petascale Computing

Project Title and PI names

Project Title: ***" Correctness Tools for Petascale Computing"***
DOE grant number: University of Maryland: DE-SC0002351

Project PI:

Jeffrey K. Hollingsworth
Computer Sciences Department
University of Maryland
College Park, Maryland 20742

Executive Summary

The purpose of this project was to develop tools and techniques to improve the ability of computational scientists to investigate and correct problems (bugs) in their programs. Specifically, the University of Maryland component of this project focused on the problems associated with the finite number of bits available in a computer to represent numeric values. In large scale scientific computation, numbers are frequently added to and multiplied with each other billions of times. Thus even small errors due to the representation of numbers can accumulate into big errors. However, using too many bits to represent a number results in additional computation, memory, and energy costs. Thus it is critical to find the right size for numbers.

This project focused on several aspects of this general problem. First, we developed a tool to look for cancelations, the catastrophic loss of precision in numbers due to the addition of two numbers whose actual values are close to each other, but whose representation in a computer is identical or nearly so. Second, we developed a suite of tools to allow programmers to identify exactly how much precision is required for each operation in their program. This tool allows programmers to both verify that enough precision is available, but more importantly find cases where extra precision could be eliminated to allow the program to use less memory, computer time, or energy. These tools use advanced binary modification techniques to allow the analysis of actual optimized code. The system, called Craft, has been applied to a number of benchmarks and real applications.

This project was the primary support of one PhD student (Mike Lam) and partially supported several additional students, and provided partial support for PI Hollingsworth. The funding has resulted in four papers, and the release of the open source tools Craft (http://sourceforge.net/projects/crafthpc/). The Craft tool has been used by others including scientists at Lawrence Livermore National Labs, Los Alamos National Labs, and by a team in New Zealand.

Detailed Project Accomplishments

Both major accomplishments of this project (cancelations and arbitrary precision) use actual runs of a program to analyze floating point behavior. To do this, our tools instrument a normal program, so that as side effect of its execution, we gather the information required for our analyses.

To instrument programs, we use the DyninstAPI library (www.dyninst.org). 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.

Cancelation Detector

To detect cancelation problems, 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 (exp1exp1 and exp2exp2) as well as the result (exprexpr). If the exponent of the result is smaller than the maximum of those of the two operands (i.e. expr<max(exp1,exp2)expr<max(exp1,exp2)), cancellation has occurred. We define the priority as max(exp1,exp2)-exprmax(exp1,exp2)-expr, 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.

In order to help programmers understand the cancelation data, 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.

Cancelation Results

To illustrate the way our tool works, consider a simple program that computers the following value: $y=1-\cos/x^2$. 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 cos0=1. This sort of example is well-known to numerical analysts, and there are many workarounds. Here it serves as an demonstration of our tool.

We wrote a simple program that evaluates this function at several points approaching x=0x=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. Figure 1 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.

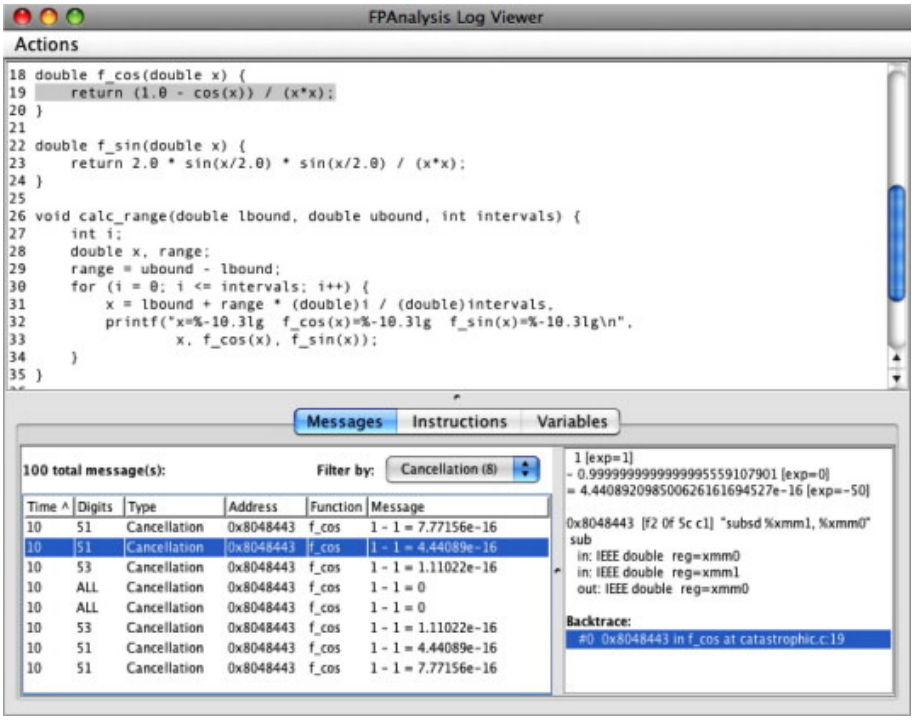


Figure 1: Screen Shot of Cancelation Detection tool.

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.

Arbitrary Precision

Our initial work on precision reduction focused on finding places where double precision floating point operations could be replaced by single precision. We introduced a comprehensive system for analyzing an application's precision requirements. Given a representative data set and a verification routine, this system builds multiple mixed-precision configurations of the application and evaluates them, choosing the one that promises the greatest benefit in terms of speedup and easing of memory bandwidth pressure. Figure 2 shows an overview of this system. A key system component is our framework for automatic binary-level mixed-precision configuration of floating-point programs.

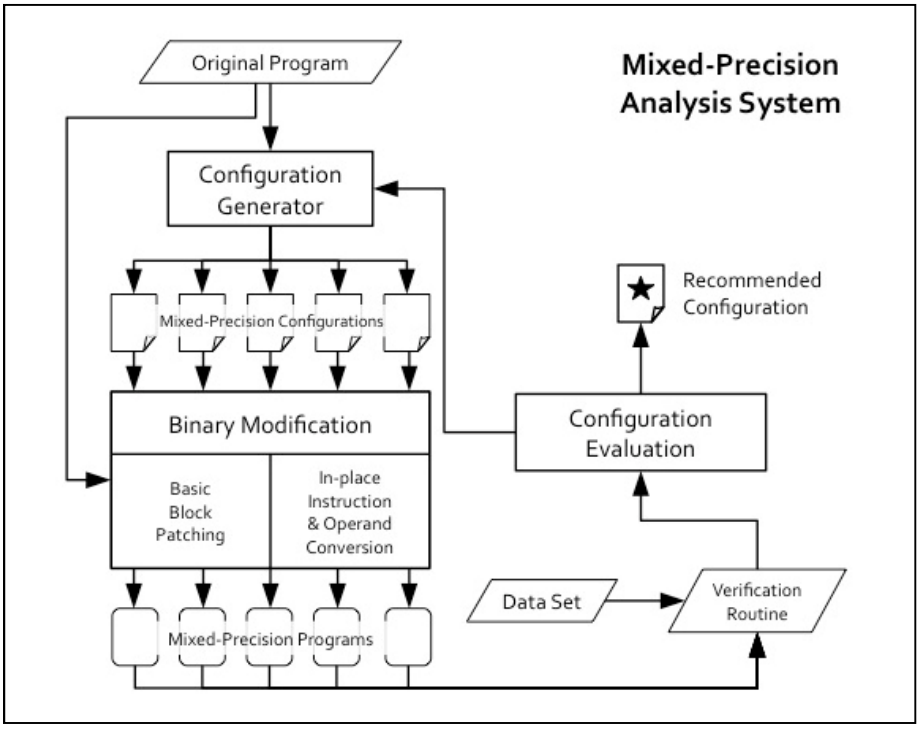


Figure 2: Overall Structure of Craft Tool.

Our tool uses a program's natural structure to find the coarsest granularity at which each part of the program can be replaced by single precision while still passing a user-provided verification routine. Our technique modifies the 64 bits that hold the double-precision representation to simulate single precision, storing a magic constant in the extra bits to indicate the presence of a modified number.

This approach does not fully realize the benefits of using single precision format but allows us to identify when single precision preserves sufficient accuracy. The source code can then be transformed to realize the full benefits for situations that we identify as safe.

The key to the ability of our tool to find the largest possible regions of code that can be replaced by single precision is our search system that automatically generates new configurations based on information learned from running from prior executions. We developed a simple automatic search system that attempts to replace as much of the program as possible using a breadth-first search through the entire program's configuration space. There are 2^n total possible configurations to test, where n is the number of floating-point instructions in the program. Since evaluating each test configuration requires a full program run, exhaustively testing every configuration is not feasible.

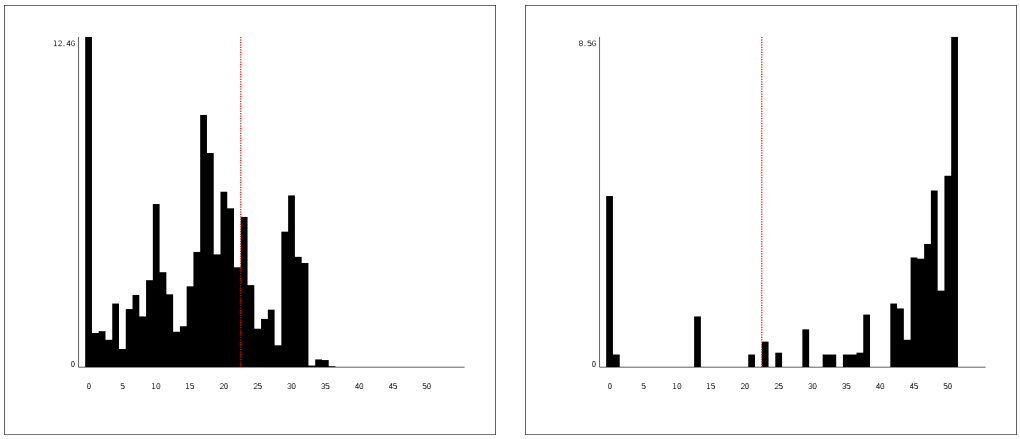
To evaluate our tool, we initially ran it on the NAS parallel benchmark suite. The results of that test are summarized in Figure 3. The second column shows the number of candidate floating point instructions that could be replaced, the third column shows the number of configurations tired, and the final two columns, the percent of static (instruction in the program) and dynamic (fraction of instructions actually executed) that our tool concluded could be replaced with single precision. The configurations tested is almost always much smaller than the candidate operations indicating that our search heuristic was working well. The percent of dynamic operations replaced (which corresponds to reduction in runtime of the program) shows that for many of the benchmarks, a large fraction of the operations could be performed in single precision.

Benchmark (name.CLASS)	Candidate Operands	Configurations Tested	Operands Replaced	
			% Static	% Dynamic
bt.A	2,342	300	98.3	**97.0**
cg.A	287	68	96.2	**71.3**
ep.A	236	59	96.2	**37.9**
ft.A	466	108	94.2	**46.2**
lu.A	1,742	104	98.5	**99.9**
mg.A	597	153	95.6	**83.4**
sp.A	1,525	1,094	94.5	**88.9**

Figure 3: Results for Single Precision Replacement Analysis for NAS Benchmarks.

The final aspect of this project was to extend the Craft tool to support detecting exactly how many bits of floating point precision a given instruction requires. This was done by adding an additional parameter, *bits*, to the configuration description for each floating point instruction. The bits parameter is used with a new variation on the instrumentation code that inserts code to truncate the result of each instruction to the number of bits in the confirmation file. Thus our tools can consider how each instruction would behave if it had a specific number of bits to represent the number.

This extension to the tool greatly expands the theoretical search space (to up 64^n where n is the number of floating point instructions in the program). However, in practice, our tool is able to prune the vast majority of these configurations and achieve acceptable performance.



(a) bt.A (b) rhodo

Figure 4: Distribution of required number of bits of precision for two programs.

In addition to providing information about specific instructions that could be replaced in the program, the arbitrary precision code allows us to visualize the distribution of the number of bits required in the program as a whole. This visualization provides quite insight into the variation in precision of different parts of the program. Figure 4, shows this distribution for two program (BY and rhodo). The red line in the center shows the number of bits of precision provided by the IEEE single precision representation. The results show that BT is mostly able to be run in single precision (but just barely). While rhoto requires almost all the bits of double precision to compute it results (with a small number of instructions requiring almost no precision).

Conclusions

This project has made tangible progress towards the goals of providing automated tools to help programmers with identifying correctness problems related to floating point precision and to provide information on how programmers can tune the level of precision to improve runtime, memory utilization, and power.

Publications

Michael O. Lam, Jeffrey K. Hollingsworth, “Fine-Grained Floating-Point Precision Analysis”, under submission to PPOPP’14.

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 International conference on supercomputing (ICS '13). ACM, New York, NY, USA.

Jeffrey K. Hollingsworth and Michael O. Lam, “Dynamic Floating-point cancellation detection”, March 2013 *Parallel Computing (39)3*.

G. W. Stewart, Jeffrey K. Hollingsworth and Michael O. Lam, “Dynamic Floating-Point Cancellation Detection”, June 2011 *First International Workshop on High-performance Infrastructure for Scalable Tools, Tuscon AZ*.