FPInst: Floating Point Error Analysis Using Dyninst

David An, Ryan Blue, Michael Lam, Scott Piper, Geoff Stoker

December 15, 2008

Abstract

Floating-point error is a well-known problem in numerical computation—it distorts results and can be very hard to measure in actual programs. Previous solutions to this problem involve increasing the precision of floating point numbers or manually inserting estimations of the error, but these solutions increase the time and memory requirements of scientific programs. Using the Dyninst framework, we have developed a prototype tool for dynamically measuring and reporting the floating point error of scientific programs. This proof-of-concept tool works on binaries at the machine code level and provides error analysis that is easy-to-use.

1 Introduction

In digital circuits, representing real numbers is a challenge. In contrast with integer representation, no simple subset of the reals will suffice; there are infinitely many real numbers just between 0 and 1. The most widely used approach is the *floating point* representation, which is analogous to scientific notation. A real number is represented as $a \cdot B^e$. The *significand*, a, and the *exponent*, e, are stored and the *base*, B, is understood as part of the representation. Reals stored in floating point representation (which we will call *floating point numbers*) require $2+log_2(a)+log_2(e)$ bits to store[1], however, these lengths are most commonly chosen ahead of time[2].

Floating point representation relies on the need for a large, but fixed amount of accuracy. Because the size of the significand is fixed, numbers can be only as accurate as the *precision*, the maximum number of digits that will fit in that size. This can be a large source of error. Figure 1 demonstrates different representation errors for a 3-digit significand and a 1-digit exponent.

[1] The extra 2 bits store the sign of both the significand and exponent.

[2] 32-bit numbers are called *floats* or *single precision*, and 64-bit numbers are called *doubles* or *double precision*.

Number	Representation	Error
100,000	$1.00 \cdot 10^5$	0
100,499	$1.00 \cdot 10^5$	499
100,500	$1.01 \cdot 10^5$	500
0.001	$1.00 \cdot 10^{-3}$	0
0.0010049	$1.00 \cdot 10^{-3}$	0.0000049
0.001005	$1.01 \cdot 10^{-3}$	0.000005

Figure 1: Examples of representation error for floating point numbers in base 10 with a 3-digit significand and a 1-digit exponent.

Another source of error is the choice of base (usually 2 because of the binary nature of computer hardware). While 0.5_{10} can be represented exactly as 0.1_2, 0.1_{10} has an infinite base 2 representation and must be rounded no matter what the available precision.

The last way error is introduced is through operations on floating point numbers. Representation error plays a large part: in the example of Figure 1, 100,000 plus anything less than 500 must be rounded to 100,000. This, in turn, introduces more error in operations by making them non-associative:

$$((250_{10} + 250_{10}) + 100,000_{10}) = 101,000_{10}$$

$$(250_{10} + (250_{10} + 100,000_{10})) = 100,000_{10}$$

Hardware choices, such as rounding, also effect the outcome (and error) of floating point arithmetic.

Still, floating point representation has proved to be both useful and flexible because the allocated space is usually sufficient for the level of accuracy that applications need. This is not the case for high performance and scientific calculations. These applications commonly require more than the usual 32-bits, using 64-bit doubles instead. This does not solve the problem, as error is still introduced, but the error is smaller because more precision is available. For applications that require still more precision, there are quad-precision (128 bits or *long double* precision) representations as well as arbitrary precision libraries. The drawback of these solutions is that increasing the size of the representation reduces the performance, as will be discussed later.

The drawbacks of larger floating point representations must be balanced against the necessary precision for a high performance application. To inform this decision, we present FPInst. FPInst estimates floating point error as it accumulates in a high performance application by automatically inserting error estimation code at every floating point instruction. The rest of this paper focuses on this tool: Section 2 discusses performance implications, Section 3 presents related publications, Section 4 explains the error analysis and accumulation calculations, Section 5 details our implementation using Dyninst, and finally, we conclude with

results and future work in Sections 6 and 7.

2 Discussion

By automatically profiling the error of an application, the developer can determine in advance not only what the current error in accuracy is, but also what the error would be if lower precision numbers were used. If you can reduce the precision of numbers in your calculations, your application may run faster. The use of single precision floating point numbers rather than double precision floating point numbers may result in performance improvements due to features of the hardware optimized to take advantage of them. Additionally, because they use fewer bits, it allows for greater throughput and storage.

Another consideration regarding performance is that in some cases higher precision is not necessary. This could be due to input data that has been measured with relatively low precision (perhaps due to poor quality instruments) Furthermore, the specific algorithms used in computations can have large impacts on the precision of the results. By taking caution when selecting the algorithms used, lower precision numbers can be used in calculations without adversely affecting the final calculation.

Some systems, including the Cray XT4, are twice as fast at performing single precision operations as double precision, and selective use of singles in applications can produce results without significant loss of accuracy [3]. The SSE extensions and 3DNow! technology on Intel and AMD processors provide SIMD operations for single precision arithmetic. An Intel processor can perform one single precision operation per cylcle, while an SSE can perform 4 flops/cycle for single precision. A more recent technology, SSE2, can perform 2 flops/cycle for double precision, but that is still half as many operations as an SSE can perform with single precision operations [14].

Some architectures can only perform operations using 32-bit floating point numbers. This is the case with many GPUs, so if one uses single precision floating point numbers, they can offload computations to these devices. Architectures like the Cell processor, GPU, and FPGA devices can provide significant performance gains when operating on single precision floating points numbers [14]. By providing estimates of precision loss one can determine if it is appropriate to translate some of the computations to lower precisions to take advantage of these devices.

As single precision numbers use half as many bits as double precision numbers, twice as many numbers can be moved in the same duration. This increased throughput is beneficial not only when copying data between computers, but especially when data is copied between cache levels within a machine, or between

devices. Additionally, because twice as many numbers can fit in the same amount of space, one should expect fewer cache misses, and fewer data movements when memory size is a restriction.

3 Related Work

The benefits of lower precision data types for performance gains has been proven useful [14]. Previous work in this area has largely focused on the numerical analysis of algorithms. A handful of tools have focused somewhat on the problem we are tackling: that of finding and analyzing floating point calculations in a general and automated fashion.

The FloatWatch tool [4], based on Valgrind, is the most similar to ours, and can analyze at run-time the data passed into floating point operations, display the range of values used by an assembly instruction, and the maximum difference that would occur between data types of different precision. This information is then displayed on a line-by-line basis associated with the original source code. The FloatWatch paper discusses the possibilities of using custom fixed-point or floating point representations, by utilizing FPGAs.

However, FloatWatch is primarily concerned with the values passed through instructions while FPInst is concerned with following values throughout the entire execution of the program. Therefore, for each memory address operated on as a floating point value, we associate an error value so that we can compile an aggregate error over the life time of a specific floating point value. By taking a data centric approach and tagging data as it moves through the program, we believe that our method can yield a more accurate calculation of the error of the final result.

Another problem with FloatWatch is a dependency on the Valgrind runtime. It would certainly be possible to build our analysis in Valgrind, but unlike tools based on Valgrind (like FloatWatch), we can both dynamically instrument running binaries, and statically instrument them by mutating binaries on disk. Static instrumentation allows for the binaries to run in environments without the need for profiling libraries.

The BitSize tool [2] statically analyzes source code to make determinations about the use of custom fixed-point and floating point representations, again with a focus on FPGA designs. Because BitSize does not execute the application, they are focusing only on the operations that are performed in the algorithms within the application. We have taken a data centric view, which we hope will provide more accurate results for applications that are run with homogeneous data sets, where all values fall in the same ranges, and the data sets are of a similar size. We believe this will be true because our tool will find areas for optimization, via the use of lower precision data types, that an analysis of the source code would not find. Additionally, we

```
// result: z, zerr
// assumes x > y
def add( x, xerr, y, yerr )
  r = x + y
  s = x - r + y + yerr + xerr
  z = r + s
  zerr = r - z + s
  return (z, zerr)
end
```

Figure 2: The addition error function.

```
add( 1.111 e 0, y = -0.001 e -1 )
  r = 1.111 + -0.0001
    = 1.1101                    ( exact )
    = 1.111 e 0                 ( rounded )
  s = 1.111 - 1.111 + -0.0001
    = -0.001 e -1               ( left to right evaluation )
  z = 1.111 e 0                 ( rounded )
  zerr = -0.001 e -1            ( leftover error )
```

Figure 3: Addition error example, `xerr` and `yerr` are zero (and not shown) for simplicity.

can find the aggregation of errors on values, which would not be possible with static analysis. An advantage of BitSize's technique, however, is that it may be faster because it does not need to execute the application to compile results.

Unfortunately, we were not able to acquire copies of either FloatWatch or BitSize to compare them properly.

4 Error Analysis

Dekker [10] presents ALGOL60 procedures and proofs of their accuracy, which built upon work by Kahan [13] and others. We derive our algorithms for error accumulation from these procedures, but with a notable change in their purpose. Dekker presents them as a method to extend the available precision—i.e., make doubles when they are not supported in hardware—but the procedures can be adapted for error accumulation because they calculate two values: the operation's single precision result, and another single precision difference between that result and the "real" result. We use these values as the result and the error. Figure 2 shows the Dekker algorithm for addition, and Figure 3 shows an example of such a calculation, starting with zero input error (`xerr` and `yerr`) for simplicity.

The idea behind these calculations is to account for the non-associative operations by making sure the operands for any calculation are as close to the same size as possible. The initial error calculation in Figure 2 works by subtracting the sum (`r`) from the larger of the two numbers and then adding the smaller of the two, so that the difference between the actual sum and the stored sum is not lost due to rounding error. Then the individual errors for each operand are added back in, with the smaller of the two first so that no error is unnecessarily lost to representation. The other operations are calculated similarly.

To account for initial representation error, we seed the error value with a function of *machine epsilon*, the bound on representation error. Machine epsilon is the unscaled maximum error between a real number and its floating point representation, which depends only on the number of digits in the significand, p, and the chosen base, B:

$$\epsilon = \frac{B}{2} \cdot B^{-p}$$

This value is available on most machines for both single and double precision representations as the symbols `FLT_EPSILON` and `DBL_EPSILON`. This ϵ-calculation assumes the scale of zero ($e = 0$), so to get the actual error bound, we must appropriately scale the initial error with the value being stored by multiplying ϵ by 2^e where e comes from the number stored [12].

Dekker's proofs of accuracy for the procedures that we use for error accumulation require assumptions about how the floating point calculations are carried out in hardware. These assumptions regard how numbers are rounded and when in the calculation they are rounded; they ensure that the hardware's calculation is as accurate as is possible for the available precision [10]. These assumptions are not met by the common x87 floating point unit that accompanies the x86 processor. The x87, according to Monniaux in [17], is not compliant to the IEEE-754 standard [20], which requires that hardware implement floating point operations as assumed by Dekker. Newer SSE floating point instructions are IEEE-754 compliant [17]. This presents a problem for our analysis that must be resolved in future work.

5 Implementation

Our current implementation performs shadow value analysis of floating point error. Because of time constraints, we currently only support x86/x87 binaries on the Linux platform.

5.1 Use of Dyninst

We use the Dyninst [7] library to insert floating point instrumentation. We chose Dyninst because (1) it works directly with the machine code and not an intermediate representation, (2) it allows us to do binary modification with minimal overhead and tool code, and (3) members of our team were already familiar with it.

Dyninst provides binary parsing and modification for both processes and images on disk, allowing us to insert our floating point analysis both dynamically and statically. The static case results in a modified binary that includes calls to our profiling library, which must be present and locatable by the dynamic linker when the program is executed.

We had to make several small additions to Dyninst that allow us to instrument all floating point instructions. We do this by tagging all instructions that begin with the following hex opcodes: `d8`, `d9`, `dc`, `dd`, and `de`. We believe this encompasses all x86/x87 instructions of interest, with the exception of instructions related to long doubles (128 bits). In the future we plan to contribute these changes back into the main Dyninst code base.

Our profiler uses this enhanced version of Dyninst to build a list of all floating point instructions and then to insert calls to our profiling library before each one. Since the logic of our analysis is relatively lengthy and time for this project was limited, we chose not to inline it in the target executable.

5.2 Our Tool

The instrumentation that Dyninst adds to the binary provides the following information to our profiling library for each executed instruction: (1) the current function, (2) the raw bytes that were relocated, (3) the number of bytes in the instruction, and (4) the current value of the various registers (these are used to locate register-relative memory operands).

We use the x86 Encoder Decoder (XED) library [1] (part of the Pin instrumentation framework [16]) to parse the bytes of the floating point instructions. We use it to obtain the operation class as well as a list of operands, their addresses, and whether they are read or written (or both). Eventually we would like to use the InstructionAPI portion of Dyninst to do this parsing, but the required features have not been completed as of the time of writing this paper.

Using the information about operations and operands, we do a shadow value analysis of floating point error. The main shadow value analysis data structure is an associative hash table that maps memory

addresses (pointers) to error values (double precision floating point numbers). The first eight entries of the table (indices 0 through 7) are used to store the error of the eight positions of the x87 floating point stack (i.e., ST(0) through ST(7)).

We handle each type of floating point instruction appropriately:

Memory Loads (FLD/FLD1/FLDZ/etc) The output register location in the shadow value table is initialized to an appropriate value, depending on the value to load. If the value is a predefined constant, the correct error is loaded directly. If the value is from a location in memory, the table is consulted to see if the value had a previously associated error. If it does, that error is copied to the new register entry. If not, the error is seeded using the value itself and knowledge about its precision (single or double).

Arithmetic (FADD/FSUB/FMUL/FDIV/etc.) Operation-specific error functions calculate a new error using the two operands and their respective errors from the shadow value table. The resulting error is stored back into the table at the appropriate location.

Memory Stores (FST/FSTP) The input register error is copied into the shadow value table for the output memory location.

The profiling library aggregates error by function, and maintains a list of entries with information about each function: (1) function name, (2) count of function calls, (3) count of floating point instructions executed, (4) count of floating point operations (excluding loads and stores), (5) maximum observed error of current run, (6) maximum observed error of all function runs, and (7) total (sum) observed error of all function runs.

At the end of a program run, the profiler prints a table containing a row for every function called during execution. Each row includes all of the above information except for the current-run metric (#5).

5.3 Other Alternatives

We explored several alternative frameworks for building our analysis tool before settling on Dyninst.

- Valgrind is a mature, dynamic binary instrumentation (DBI) framework for building debugging and analysis tools[19]. The execution involves a tool invocation, loading of a program of interest, and a just-in-time recompilation of that program's machine code one block at a time. The Valgrind core disassembles the code of interest into an intermediate representation. The tool plug-in provides instrumentation and then the core reconverts this into machine code. Valgrind is well suited for all

types of instrumentation and may offer more features than other frameworks; however, it comes at the price of additional overhead and slower performance. Floating point instruction handling is also a bit more rudimentary than the handling of general-purpose registers and instructions. Each floating point instruction is encoded in a single UCode instruction that hides the original operand and the floating point register values [18]. Thus, there is no built-in support for tools to shadow floating point registers.

- Low Level Virtual Machine (LLVM) is a compiler infrastructure that enables program optimizations during compile-time, link-time, run-time, and/or idle-time [15]. It supports various languages including C and C++ by translating them into an intermediate representation (IR) form using the front-end similar to the GNU C Compiler (GCC). LLVM provides a framework for writing a custom virtual machine that is capable of doing compiler optimizations on the IR. Using this framework, it is conceivable to design a compiler that instruments floating point operations on the IR. Unfortunately, the IR language does not truly reflect the behavior of the actual x86 machine code. This is due to a number of reasons, but two major ones are (1) its usage of static single assignment (SSA) form to represent registers and (2) various optimizations that may occur at any point of program compilation and execution. SSA form basically allows an infinite number of registers, facilitating optimizations without the nuisance of register allocations and spills. However, this is undesirable for instrumenting all floating point instructions because many load and store operations are disregarded in the IR. Along with this problem, compiler optimizations that happen after instrumentation may distort our results because the final machine code does not reflect the behavior we have observed with the IR. Although we could have designed a custom virtual machine with these features turned off and performed our analysis strictly on the IR, we decided not to further investigate LLVM since Dyninst appeared to have features we needed and is capable of binary code translation.

- Pin is another actively developed and supported tool for dynamic instrumentation [16]. Support is primarily directed at Linux binary executables for various Intel processors, but it also supports instrumentation of Windows programs on some Intel processors. Pin instruments by adding code dynamically while the executable is running and can either start execution with a program of interest or be attached to an already running process. The Pin framework is well-suited for most common instrumentation requirements, though its lack of built-in support for certain features, like providing shadow memory, may preclude more heavyweight analysis. None of the members of our team had any experience with Pin.

- DynamoRio is a runtime code manipulation system that was jointly created by researches at Hewlett-Packard and MIT [6]. Dynamo was built by HP as a dynamic optimization prototype capable of optimizing a native program binary at runtime and forms the base onto which RIO (Runtime Introspection and Optimization) was added. The collaborative tool was built for speed and provides an exact representation of the target program code which lends to being a very good choice for lightweight binary instrumentation. The system allows modification of the runtime code stream for changing, adding, or removing individual instructions. This is a somewhat different approach to instrumentation from Dyninst's trampolining technique. It also lacks support for more heavyweight instrumentation like shadow values.

- The Dynamic Execution Layer Interface (DELI) also came from Hewlett-Packard's Dynamo research effort [11]. It was planned as an infrastructure for dynamic code transformation tools and provides a layer of hardware abstraction. DELI uses an intermediate representation and also provides virtual registers. Intended uses were dynamic code patching, decompression, and decryption. The DELI code does not seem readily available and may no longer be actively developed.

- Strata is an infrastructure for building software dynamic translators specifically design driven to be extensible and retargetable [21]. It uses standard dynamic binary compilation and caching. Strata modifies the target program code as minimally as possible. Development targeted SPARC and MIPS (Solaris and IRIX), and while the source code is supposedly available, it is not easily locatable indicating that current development may be stalled or no longer in the public domain.

6 Results

Figures 4 and 5 contain the code and analysis results for a simple contrived example containing various incarnations of individual floating point instructions. Figure 6 shows the result of running our analysis on the whetstone benchmark [9].

The columns represent (1) the function name, (2) the number of times that function was called during execution, the number of floating point (3) instructions and (4) operations encountered during execution, (5) the maximum error accumulated inside the function, and (6) the sum error across all runs of the function. The instruction count (FIns) includes memory loads and stores while the operation count (FOps) does not.

To justify our results, we used PAPI [5] to count total floating point operations executed for the two

examples. Calls to the PAPI library functions were manually instrumented around calls to functions of our interest. The results were identical to Figures 5 and 6 when we used GCC 3. However, there were differences when we used GCC 4 because the compiler performed optimizations that converted floating point instructions into multiple integer instructions, which our tool does not currently handle.

We also tried to verify the actual error results of our tool. Figures 7 and 8 contain the code and analysis results of a simple error accumulation example. The Real Error is measured by taking the value at the end and subtracting from it the expected value (known *a priori*). The discrepancy between the Max Error and Real Error shows that we are still significantly underestimating the amount of error in this program. We suspect that this may be because some of our initial assumptions about x86/x87 are not true (see Section 4 for a discussion of this).

```
float ffunc() {
    float a = 42.7f;
    float b = 3.14f;
    return a + b;
}

float ffunc2() {
    float a = 42.7f;
    float b = 3.14f;
    float c = 2.5f;
    return (a + b) * c;
}

int main() {
    ffunc();
    dfunc();
    ffunc2();
    for (i=0; i<3; i++)
        ffunc3();
    mfunc();
}
```

```
float ffunc3() {
    float a = 42.7f;
    float b = 3.14f;
    return a * b;
}

double dfunc() {
    double a = 42.7;
    double b = 3.14;
    return a + b;
}

double mfunc() {
    float a = 42.7f;
    double b = 42.7;
    return a + b;
}
```

Figure 4: fptest1 Code

7 Future Work

We have many ideas and possible directions for future work:

- Do more extensive testing and evaluation to verify the correctness of our current analysis. Include comparisons with other similar tools such as FloatWatch and BitSize. Apply manual numerical analysis

```
================================================================================
   Function name          Calls   FIns  FOps            Max            Sum
================================================================================
   dfunc                      1      6     1     1.5099e-14     1.5099e-14
   ffunc                      1      2     1    8.10623e-06    8.10623e-06
   ffunc2                     1      3     2    8.58307e-06    8.58307e-06
   ffunc3                     3      6     3    8.10623e-06    2.43187e-05
   mfunc                      1      4     1    7.62939e-06    7.62939e-06
```

Figure 5: fptest1 Results

```
Loops: 1000, Iterations: 1, Duration: 70 sec.
C Converted Double Precision Whetstones: 1.4 MIPS

================================================================================
   Function name          Calls     FIns     FOps            Max            Sum
================================================================================
   P0                    616000        0        0              0              0
   P3                    899000 10788000 5394000    3.44169e-15    3.09408e-09
   PA                     14000  2016000 1344000    8.71525e-15    1.02866e-10
```

Figure 6: Whetstone Benchmark Results

to verify the accuracy of our results.

- Contribute any changes to the Dyninst API back into the main code base.
- Optimize the current analysis and inline as much of it as possible into the rewritten executable to avoid function call overheads during execution.
- Extend our analysis to support other platforms and floating point representations and instruction sets (ex. long doubles, SSE and x86_64).
- Include more methods of error estimation and allow for comparisons between methods.
- Precisely enumerate all assumptions made by the analysis, and verify that they are valid.
- Allow users to seed errors with more scientifically-appropriate values, such as those obtained from measurement bias.
- Provide a graphic user interface that programmers can interact with to learn about their programs more quickly and precisely, or integrate reporting into an integrated development environment.
- Include more shadow value information, such as maximum and minimum values that data takes on, to

```
Sum to 1 (excerpt):

    sum = 0.0;
    for (i=0; i<10000; i++) {
      sum = sum + 0.0001;
    }

Sum to 1000 (excerpt):

    sum = 0.0;
    for (i=0; i<10000000; i++) {
      sum = sum + 0.0001;
    }
```

Figure 7: sum0001 Code

```
Output:

The correct answer would be 1.0000
            float(size=4) = 1.000054
           double(size=8) = 0.999999999999906186

The correct answer would be 1000.0000
            float(size=4) = 1087.724243
           double(size=8) =  999.999999820615471
```

Test	Data type	FIns	FOps	Sum Error	Real Error
Sum to 1	float	4000	1000	1.19e-07	5.40e-05
	double	4000	1000	5.95e-16	9.38e-14
Sum to 1000	float	40000000	10000000	1.19e-07	87.7
	double	40000000	10000000	8.25e-11	1.79e-07

Figure 8: sum0001 Results

provide insight into the possibility of choosing different fixed-point or floating point representations to use.

- Improve analysis to track floating point values and errors through integer registers, memory copying (ex. memcpy), and memory resets (ex. memset). This is particularly a problem with GCC 4, which tries to optimize floating point parameter passing by moving floating point values on and off the stack using multiple integer move instructions rather than an FLD or FST.

- Explore whether we can do better analysis by inserting instrumentation *after* each instruction as well as before it–perhaps we can get a more precise estimation of an individual calculation's error this way.

If so, we should examine the speed trade-off.

- Improve reporting of our analysis results by different aggregation. An obvious improvement is to provide a map that associates stack/memory locations to source-level variable names at any given instruction. This will help programmers debug programs on source level, not just on the binary instruction level. We might also provide aggregation by line of code to help pointpoint partcularly troublesome areas.

- Implement program slicing [22] to help programmers understand code more effectively and precisely by analyzing only the code that affects (or is affected by) a point of interest. It can provide richer information about how a floating point instruction might affect (or be affected by) another instructions. Binary executable slicing [8], in particular, applies slicing to the binary instruction level rather than source code level. In addition to reporting, slicing can help reduce the number of instrumentations by computing the code relevant to the point of interest and instrumenting only those instructions. This will minimize the overhead caused by the instrumentation.

8 Conclusion

We have developed a tool to assist in the analysis of floating point error. Although more work is still necessary for publishable results, the preliminary results are promising and indicate that we have developed a tool that can be useful for not only giving developers a better understanding of their applications for improving the accuracy of their results, but for assisting them in making decisions that may lead to improved performance.

References

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

[2] A. Abdul Gaffar, O. Mencer, W. Luk, and P. Cheung. Unifying bit-width optimisation for fixed-point and floating-point designs.

[3] G. Alvarez, M. Summers, D. Maxwell, M. Eisenbach, J. Meredith, J. Larkin, J. Levesque, T. Maier, P. Kent, E. D'Azevedo, et al. New algorithm to enable 400+ TFlop/s sustained performance in simulations of disorder effects in high-T c superconductors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press Piscataway, NJ, USA, 2008.

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

[5] S. Browne, C. Deane, G. Ho, and P. Mucci. Papi: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, June 1999.

[6] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* PhD thesis, Massachusetts Institute of Technology, 2004.

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

[8] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. *Software Maintenance, 1997. Proceedings., International Conference on*, pages 188–195, Oct 1997.

[9] H. Curnow and B. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, 1976.

[10] T. J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[11] G.Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. Fisher. Deli: A new run-time control point. November 2002.

[12] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23:5–48, 1991.

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

[14] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. Tools and techniques for performance—Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM Press New York, NY, USA, 2006.

[15] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, March 2004.

[16] 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.

[17] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):1–41, 2008.

[18] N. Nethercote. *Dynamic Binary Analysis and Instrumentation.* PhD thesis, University of Cambridge, November 2004.

[19] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. volume 42, pages 89–100. PLDI, ACM, 2007.

[20] I. T. P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic.* IEEE, New York, Aug. 12 1985.

[21] K. Scott and J. Davidson. Strata: A software dynamic translation infrastructure. Workshop on Binary Translation. IEEE, July 2001.

[22] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.