Floating-Point Shadow Value Analysis

Michael O. Lam
Department of Computer Science
James Madison University
Harrisonburg, Virginia 22807
Email: lam2mo@jmu.edu

Barry L. Rountree
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, California 94550
Email: rountree4@llnl.gov

***Abstract*—Real-valued arithmetic has a fundamental impact on the performance and accuracy of scientific computation. As scientific application developers prepare their applications for exascale computing, many are investigating the possibility of using either lower precision (for better performance) or higher precision (for more accuracy). However, exploring alternative representations often requires significant code revision. We present a novel program analysis technique that emulates execution with alternative real number implementations at the binary level. We also present a Pin-based implementation of this technique that supports x86_64 programs and a variety of alternative representations.**

I. Introduction

Since its standardization in 1985, IEEE floating-point arithmetic [1] has become the primary implementation of real-valued arithmetic on most computing platforms. Although it has well-known weaknesses (e.g., rounding error and cancellation [2]), it is widely used in scientific computing because it provides a wide dynamic range and can be implemented efficiently in hardware. As high-performance computing continues to scale to exascale and beyond, concerns regarding the precision, memory bandwidth, and energy usage of IEEE floating-point arithmetic will become increasingly important [3]. Alternatives exist, but porting existing programs is time-consuming, with no guarantee that the results will be useful.

We provide a software framework for emulating alternative real-numbered representations without modifying the original program. We use binary instrumentation to insert new code that runs alongside the original code, performing operations on a copy of the program's data. To differentiate the inserted code and data from the originals, we refer to the inserted code as *shadow code* and to the copied data as *shadow values*. We call the general technique *shadow value analysis*.

We do our analysis at the binary level because it most accurately reflects the runtime behavior of the original program. Digital real-valued arithmetic (including IEEE floating-point) is often non-associative and non-distributive, so the compiler must preserve the semantics of the original program when generating the specific operations that take place at runtime. A routine could be more or less accurate depending on which instructions are chosen and how they are scheduled. Source-based analyses such as Rose or LLVM will miss these effects. Recent work in compiler verification highlights incorrect compiler transformations that decrease floating-point accuracy [4].

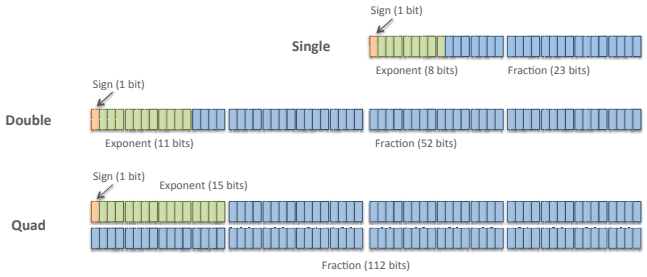

Fig. 1. IEEE standard floating-point formats

Figure 1 shows a graphical representation of three IEEE floating-point representations. Within a fixed-sized bit field, there are three parts: 1) the sign bit, 2) a biased exponent, and 3) a fractional part (sometimes called the *significand* or *mantissa*) with an implicit leading digit. The figure shows these bit field layouts for three different sizes. In each case, the value stored is $(-1)^{sign} \cdot 1.frac \cdot 2^{exp}$; the only difference is the number of bits designated for the fraction and exponent. Using more bits provides more precision, but increases storage costs and computation time.

Most scientific programs are written to use 64-bit double-precision IEEE arithmetic. Our framework allows developers to experiment with lower or higher precisions without having to port their code manually. It also allows them to experiment with different representations entirely.

One such alternative representation is John Gustafson's *universal number* scheme [5], which is a variation on floating-point numbers. In this scheme, the fixed-width fraction and exponent fields are replaced by variable-sized fields, and new fraction-size and exponent-size fields are added to track the sizes at runtime. The widths of these new size fields is fixed by the *environment*. For example, the environment (4,6) uses four bits to store the size of the exponent and six bits to store the size of the fraction. The format also includes a *ubit* flag that indicates whether the represented quantity is exact or whether it lies in the space between the given floating-point number and the next representable value. These numbers are sometimes called *unums*.

Figure 2 shows a graphical representation of several unum environments. This representation quantifies the error introduced by rounding and magnified by repeated computation.

978-1-5090-3918-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ESPT.2016.10

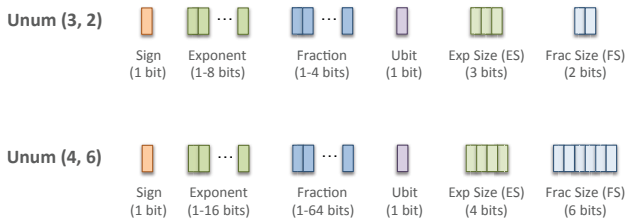

Fig. 2. Gustafson's universal number formats

```
double sum = 0.0;
for (int i = 0; i < 10; i++) {
    sum += 0.1;
}
printf("%25.20f\n", sum);
```

Fig. 3. Sample C program

With current implementations, it is infeasible to use unums for production-level computation because of the added storage and computation costs. However, many scientific developers are interested in investigating how their numerical programs behave with unum arithmetic. This is because unums explicitly encode error while standard IEEE floating-point representations do not. Our framework provides an automated way for these developers to try unums with minimal time investment.

For example, consider the simple C program in Figure 3. It performs ten additions of the quantity 0.1 to an accumulator variable. If done in infinite precision with no rounding error, the result should be exactly 1.0. However, 0.1 cannot be represented exactly in a binary representation such as IEEE floating-point doubles, and the resulting rounding error accumulates across the multiple operations. The actual program output is 0.99999999999999988898, which is accurate to about fifteen decimal places.

Figure 4 shows an excerpt from the compiled assembly code for this program alongside pseudocode representing the new shadow code that our framework inserts. The new code maintains shadow values for the two XMM registers in one table (`xmm`), and the final result is stored to another table (`mem`) when it is written back to memory.

Table I shows the results of running our analysis using five different representations: the standard IEEE single and double precisions, a 128-bit precision simulation using the GNU MPFR library [6], and three different unum simulations with varying environments using a C implementation of unums. The table reports the final shadow value as well as the relative error of the shadow value based on the original double-precision version. As expected, the double-precision version is identical to the original and the single-precision version is less accurate. The 128-bit precision result, however, is not significantly more accurate than the original result, highlighting that the accuracy of the computation is constrained by the conversion of the constant 0.1. Because this conversion happens at compile time, it is not possible to achieve full 128-bit accuracy at runtime unless the original constant is also initialized with that precision during compilation. The unum results show that a larger environment leads to a tighter bound on the answer and a correspondingly lower relative error.

The rest of the paper is arranged as follows. In Section II we describe our techniques in detail, discussing many of the problems that we encountered during development and our solutions to these problems. In Section III we present preliminary results, including our overheads on several benchmarks and proxy applications. In Section IV we discuss related work and differentiate our contributions from previous work. Finally, we conclude in Section VI.

II. Methods

We implemented our techniques using Intel Pin [7], a binary instrumentation framework that provides just-in-time insertion of analysis code into a running program. Using Pin, we insert code that performs all of the original instructions on alternative values, which we call "shadow" values because they have tight, one-to-one correspondence with the original values. For now, our analysis only supports shadow value analysis on 64-bit floating-point values, as this is the most common data type in scientific communication. We used version 2.14 of Pin for our analysis, because we found that the newest version of Pin (3.0) contained a regression that prevented us from linking against third-party libraries (such as GNU MPFR).

A. Implementation

Our primary instrumentation inserts code after every SSE (Streaming SIMD Extension) instruction in the target program, including both scalar and packed vector instructions. The inserted code includes calls to analysis functions that do the corresponding operations on the shadow values that are associated with the original operand locations. Pin optimizes the analysis code, inlining as much as possible to improve runtime performance.

For example, consider the SSE instruction "`addsd xmm0, 0x400def`", which adds the double-precision value located at address `0x400def` to the double-precision value located in register XMM0. For this instruction, our analysis inserts code that retrieves the shadow value associated with XMM0 as well as the shadow value associated with the memory location `0x400def`, adding them together and saving the result as the new shadow value associated with the register XMM0. The shadow values could be alternative precisions (e.g., single precision or quad precision) or they could be entirely different numerical representations (e.g., unums or rational numbers).

We maintain two major data structures for shadow values. Both structures associate shadow values with system locations. The first structure stores shadow values for the SSE registers, and the second structure stores shadow values for memory locations. The register shadow value data structure is a simple array with four slots per register because the original registers

```
Original machine code:                                  Inserted shadow code:

  pxor   xmm0, xmm0          (set to 0.0)               xmm[0] = convert(0.0)
  mov    eax, 10
  movsd  xmm1, 0x400628      (load 0.1)                 xmm[1] = convert(*(0x400628))
loop:
  sub    eax, 1
  addsd  xmm0, xmm1          (increment)                xmm[0] += xmm[1]
  jne    loop
  movsd  0x8(rsp), xmm0      (store sum)                mem[rsp+0x8] = xmm[0]
```

Fig. 4. Compiled assembly of program from Figure 3

Shadow Value Type	Exp Size	Frac Size	Final Shadow Value	Relative Error
32-bit (native single)	8	23	1.000000	1.19e-07
64-bit (native double)	11	52	1.000000000000000	0
128-bit GNU MPFR	15	112	1.00000000000000005551e+00	1.11e-16
Unum (3,2)	8	4	(0.9375, 1.1875)	0.059
Unum (3,4)	8	16	(0.9999847412109375, 1.0000457763671875)	1.53e-05
Unum (4,6)	16	64	1.0000000000000000005551...182	1.11e-16

TABLE I
ANALYSIS RESULTS ON SAMPLE PROGRAM

are 128 bits wide, which is large enough to store four 32-bit floating-point numbers.

The memory shadow value data structure is more complex. We have two alternative implementations: an STL unordered map and a simple array with bit mask look-ups. The STL map implementation is cleaner but slower. The array-based implementation is theoretically prone to collisions, but in practice we avoid this by making the array very large, delegating page management to the operating system. We use the array-based implementation for all overhead results in Section III.

Static memory is the easiest to track because it is not allocated or de-allocated during runtime. Static shadow values are usually reported at the end of the `main` function, although this behavior is customizable. Tracking heap shadow values requires us to instrument `malloc` and `free` and related functions to maintain a mapping of valid heap locations. This prevents us from re-using old shadow values for newly-allocated regions of memory. Any heap shadow values that are still allocated at the end of `main` will be automatically reported alongside the static memory shadow values. Finally, we track shadow values for locations on the system stack but do not automatically report them because they are often invalidated when stack frames are destroyed.

The user can request the shadow value for any memory location at any time during execution by linking their application with our runtime library and adding explicit calls to our analysis functions. Users can request shadow value reports for single variable or for an entire array at a time. For an additional overhead cost, our analysis can also monitor and report high relative error at runtime, which is useful for determining the earliest point at which shadow calculations diverge significantly from the original calculations.

B. Issues Encountered

This section summarizes the major issues encountered while implementing our shadow value analysis techniques.

1) **Tracking floating-point values through memory**—Floating-point values are sometimes stored in regions of memory that are not explicitly designated as floating-point data, and they are sometimes manipulated using instructions that are not part of the SSE instruction set. This can be difficult or impossible to detect even with static analysis because of variable aliasing. This analysis is especially difficult for C and C++ programs because of pointers. To avoid missing such data movement, we instrument all memory reads and writes even if the instruction is not an SSE floating-point move instruction (i.e., `movsd`). For memory reads, we check to see if the effective address is already associated with a shadow value; if it is not, we assume this is a new floating-point value (i.e., a compiled constant or a value read from an I/O buffer) and initialize a new shadow value. For memory writes, we again check to see if the effective address is already associated with a shadow value; if it is, we clear the old shadow value and replace it with the shadow value associated with the newly-written value. The latter is not necessarily a problem, as data buffers are often reused. However, it could also be indicative of a bug in the analysis, so we do provide the option to log such incidents if the user wishes to do so.
2) **Lack of floating-point semantics**—Intel Pin provides a robust interface for querying attributes of instructions that are candidates for instrumentation; however, it does not provide enough semantic information for

our analysis. For instance, it provides information about which registers are accessed, but does not include information about whether the register is being used as a scalar or a packed register (e.g., `addsd` vs. `addpd` instructions) or the specifics of complex intra- and inter-register data movement (e.g., `shufpd`). To ensure that our analysis captures all computation and data movement, we encoded these semantics manually. For packed instructions, we hard-code analysis routines that do multiple shadow value operations, and for complex data movement instructions we mirror the movement in our shadow value tables. We provide helper methods and macros to streamline the addition of semantics for new instruction sets in the future. In addition, we found that compilers often emit specialized floating-point bitwise idioms, usually manipulating the sign bit to negate or take an absolute value. For these instructions, we examine the register contents at runtime to detect these patterns and manually do the corresponding operation on shadow values. Table II summarizes the major idioms we support.

3) **Verifying correctness**—To check for correctness, we implemented an analysis mode that uses the IEEE double-precision representation, essentially duplicating the original computation in shadow values. We also provide an option to do online consistency checks during runtime. This will flag any shadow value that is not bit-for-bit identical to the original value. Values are checked after all memory writes or (optionally) after every single instruction, although the overhead is significantly higher for the latter. In addition, we have a variety of internal consistency checks and warnings that will trigger and notify the user whenever an event happens that could be indicative of incomplete analysis (i.e., an unhandled instruction opcode).
4) **Efficiency of analysis**—Instrumenting a binary will always cause some runtime overhead, and in our case the analysis requires the insertion of significant amounts of new code. In initial tests, runtime overheads were often over 1000X. To improve these rates, we perform as much analysis as possible at the point at which the instrumentation is inserted. This will still add runtime overhead because Intel Pin is a just-in-time instrumentation framework, but the costs are amortized over an execution because Pin caches instrumentation between invocations. To further reduce overhead, we added many individual per-opcode analysis handlers to minimize branching. We also took advantage of Pin's built-in "If/Then" instrumentation calls to minimize the performance degradation of branching when it is unavoidable (i.e., checking shadow values to see if they are initialized).
5) **Information overload during reporting**—For completeness, we by default report all valid shadow values at the end of the `main` function. While this is comprehensive, it often results in information overload. To address this challenge, we provide several options for restricting output. The first option is an error threshold for most analysis types that causes the reporting system to suppress the output of any shadow value that does not exceed the error threshold, highlighting the values that are likely to be of interest to the user. The second option allows the user to specify a different function as the primary reporting point. The third option allows the user to request shadow value reporting at an arbitrary point during execution. With the latter, the user can then disable the default reporting and only receive shadow value information for the particular data structures of interest.
6) **MPI communication**—To be comprehensive, our analysis must intercept double-precision MPI communication in the original program and augmented this communication to include the corresponding shadow values. This poses a problem for some types of analysis that dynamically allocate parts of their shadow value (e.g., the GNU MPFR library). Thus, we added packing and unpacking routines that must be implemented for each shadow value type to prepare the shadow values for transmission as raw bytes. This also meant that the MPI data type for the communication must change from `MPI_DOUBLE` to `MPI_BYTE`, which could affect the underlying message pattern in optimized MPI implementations. Including shadow values in MPI collectives also requires special handlers for reduction operations, which we implemented using user-defined reductions. Our custom reductions do the original reduction operation in addition to unpacking the shadow values and performing the corresponding reduction operation before re-packing them for further communication. Our analysis does not currently support MPI custom data types that contain floating-point values.

III. Results and Discussion

Table III shows overhead results on the floating-point benchmarks from the serial NAS Parallel Benchmark suite [8], compiled with the GNU compiler version 4.9.2 with -O3 enabled. Table IV shows selected results from the same set of benchmarks compiled with the Intel compiler version 16.0.3 with -O3 enabled. Experiments were run on Xeon E5-2695 2.4GHz cluster nodes w/ 128GB RAM. Each individual cell value is the minimum of twenty individual runs, to avoid runtime variation due to operating system effects.

The first column contains the name and the size of the benchmark in increasing order (W, A, C). The second column contains the original runtime in seconds for reference. The remaining columns contain the overheads for each shadow value type as a slowdown factor based on the times in the second column. The "Native32" analysis represents simulating single-precision arithmetic, the "Native64" analysis represents the online verification mode (including memory-write checking), and the "MPFR128" analysis emulates 128-bit precision using the GNU MPFR library. The "Baseline" column is a

General Form	Meaning
`AND [xmmX], 0x0`	set to zero
`AND [xmmX], 0x7fffffffffffffff`	clear sign bit (take absolute value)
`OR [xmmX], 0x0000000000000000`	copy a value
`XOR [xmmX], [xmmX]`	set to zero
`XOR [xmmX], 0x8000000000000000`	invert sign bit (negate)

TABLE II
BITWISE FLOATING-POINT IDIOMS

Benchmark	Original (s)	Baseline (X)	Native32 (X)	Native64 (X)	MPFR128 (X)
bt.W	2.2	23	46	60	247
bt.A	50.1	21	41	55	237
bt.C	926.8	19	38	50	
cg.W	0.2	10	12	13	133
cg.A	0.9	8	11	12	126
cg.C	248.7	3	7	5	55
ep.W	1.9	10	12	15	74
ep.A	14.9	10	12	15	74
ep.C	239.1	10	12	15	74
ft.W	0.2	17	23	28	171
ft.A	3.5	11	17	24	146
lu.W	4.7	28	41	46	233
lu.A	32.1	28	40	45	288
lu.C	601.7	26	37	41	
mg.W	0.2	32	38	37	291
mg.A	1.2	32	38	37	308
sp.W	4.6	15	21	29	195
sp.A	30.7	13	20	26	185
sp.C	587.3	12	18	24	
ua.W	2.0	34	43	49	285
ua.A	16.4	31	39	45	290
ua.C	387.4	21	28	32	214
MIN		3	7	5	55
AVG		19	27	32	191
MAX		34	46	60	308

TABLE III
OVERHEADS FOR SERIAL NAS PARALLEL BENCHMARKS (GNU COMPILER)

special mode that inserts blank shadow value instrumentation, providing a lower bound on the overhead imposed by Intel Pin and our framework. Some sizes are missing; in these cases the benchmark was not provided in that size. Additionally, some of the MPFR128 results are missing; these tests did not complete in the 24 hour maximum time limit on our testing cluster.

Overall, the baseline overheads average around 20X, single-precision simulation overheads average around 30X, and 128-bit precision simulation overheads average around 200–250X. The overheads for GNU MPFR are significantly higher than the other modes because shadow value operations consist of calls to the library, rather than individual machine code instructions.

These overheads show the feasibility of our techniques, although the overheads are still high enough to prevent analysis on full-scale production runs. The overheads are higher for the benchmarks compiled with Intel because they are more efficient and thus take a harder hit under our instrumentation, which disrupts memory caching. The overhead does tend to stay the same or decrease as the problem size increases for a particular benchmark, showing that the cost of inserting analysis does amortize and will not grow to dominate the original application's runtime. We suspect that future optimization efforts could reduce the overhead further.

Table V contains overhead results for the serial version of the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) proxy application [9]. Table VI contains overhead results for the MPI version of the same application, scaling both the problem size (the "s" parameter) and the number of MPI processes. For the MPI experiments,

Benchmark	Original (s)	Baseline (X)	Native32 (X)	Native64 (X)	MPFR128 (X)
bt.A	41.5	20	40	55	307
bt.C	805.0	18	35	48	
cg.A	0.8	6	10	11	139
cg.C	259.0	2	4	5	57
ep.A	6.9	20	25	28	194
ep.C	109.8	20	25	28	193
lu.A	34.7	9	20	27	214
lu.C	660.8	8	18	25	
sp.A	25.6	23	30	34	274
sp.C	510.6	19	26	29	
ua.A	14.2	36	49	51	322
ua.C	329.9	29	38	40	232
MIN		2	4	5	57
AVG		19	29	34	243
MAX		36	49	62	330

TABLE IV
OVERHEADS FOR SERIAL NAS PARALLEL BENCHMARKS (INTEL COMPILER)

we used MVAPICH 1.2 with one process per node and we report the minimum of ten individual trials. Although the per-trial variation was higher than the variation in the serial results, the average overheads trended towards similar magnitudes as the experiments scaled to larger sizes and processor counts, and in fact in some cases the overhead was noticeably lower.

We also present preliminary shadow value error results for the serial version of LULESH. For these experiments, we examined the shadow values for "reportable" relative errors (with the threshold arbitrarily chosen to be 1%) at the end of execution. Table VII shows the value counts (second column) and percentages of those values with reportable relative errors, using both Native32 and MPFR128 shadow values. There were of course zero errors for Native64 because the results were identical–thus, we do not include those results. For Native32, the error is calculated relative to the original double-precision value (assumed to be more accurate), while for MPFR128 the error is calculated relative to the shadow value. This means that we always use the higher-precision result as the "true" value for the purposes of relative error calculation.

In general, the error rates for Native32 are much higher, as expected. In addition, the error rates tend to be increasing, showing that even as the number of results increases, the errors seem to be expanding. This matches the general behavior of forwards error propogation. Conversely, the MPFR128 error rates seem to be relatively stable, showing the lower variability between double precision and MPFR-simulated 128-bit precision. However, the rates are still non-zero indicating that perhaps the use of double precision is still having significant rounding error on at least a few of the calculated quantities. We plan to investigate this further in future work.

We also performed some preliminary unum-based shadow value analysis on LULESH. Unfortunately, many of the shadow values diverged to NaN (not-a-number) for any non-trivial simulation size. The divergence matches the behavior observed by others who have written a manual unum port of LULESH. This often happens when an unum's range expands to include zero, which will produce a NaN if the unum is used as the denominator of a division operation. The authors of the manual port currently solve this problem by periodically collapsing the unum to its discrete midpoint value at manually-chosen points during execution. Unfortunately, it is unclear how to determine these points using automated analysis.

IV. RELATED WORK

Backwards and forwards error analyses are well-known numerical techniques for quantifying upper bounds on rounding error [10], [11]. Forwards analysis begins with a bound on input error and observes its growth during calculation, while backwards analysis begins with the output values and determines how much variation in the inputs could be accounted for by the intermediate calculations. Automated approaches to such error analyses are usually based on static analysis techniques, highlighting program areas where precision loss could be a problem and suggesting program transformations to reduce the loss [12], [13]. However, these tools often conservatively overestimate the amount of error encountered during a typical run of an algorithm. Recent work in this area includes automated verification of compiler optimizations and implementations of transcendental functions, techniques that nicely complement our work by highlighting some of the incorrect compiler transformations and floating-point libraries that impact numerical accuracy [4], [14].

Source-to-source translation is another static analysis approach that resembles ours, rewriting programs at the source or intermediate representation level. This includes tool frameworks such as Rose [15] or LLVM [16]. To our knowledge, there is no such framework specifically for floating-point arithmetic and generalized real-numbered arithmetic. Additionally, these techniques do not generally incorporate

Size	Original (s)	Baseline (X)	Native32 (X)	Native64 (X)	MPFR128 (X)
s=10	0.2	28	42	46	192
s=20	5.6	25	36	40	170
s=30	29.0	24	36	40	177
s=40	95.7	22	34	37	167
s=50	241.4	22	34	37	173
MIN		22	34	37	167
AVG		24	36	40	176
MAX		28	42	46	192

TABLE V
OVERHEADS FOR LULESH PROXY APPLICATION (SERIAL)

Size and MPI Proc Count	Original (s)	Baseline (X)	Native32 (X)	Native64 (X)	MPFR128 (X)
s=10 np=1	0.2	28	43	49	212
s=10 np=8	0.6	54	60	69	244
s=10 np=27	1.2	76	93	93	246
s=20 np=1	5.5	24	36	40	181
s=20 np=8	12.8	25	36	40	204
s=20 np=27	19.8	27	39	44	219
s=30 np=1	29.1	23	35	40	182
s=30 np=8	65.3	23	36	40	
s=30 np=27	103.3	23	36	40	
MIN		23	33	40	181
AVG		44	50	57	219
MAX		156	117	124	264

TABLE VI
OVERHEADS FOR LULESH PROXY APPLICATION (MPI)

Size	Shadow Values	Native32 (err%)	MPFR128 (err%)
s=10	37,402	1.14	0.02
s=20	280,492	22.08	0.01
s=30	927,382	24.45	0.00
s=40	2,176,072	37.08	0.01

TABLE VII
REPORTABLE SHADOW VALUE ERROR RATES FOR LULESH PROXY APPLICATION

compiler optimization effects because they general work at or above the abstract syntax tree transformation level.

Our approach is more similar to dynamic analyses that instrument a compiled program at runtime and are therefore more sensitive to compiler optimizations and data set differences, as well as being less prone to worst-case error results. Previous work in this area includes a technique for online cancellation detection [17], an automated mixed precision search framework [18], and a truncation-based general precision analysis tool [19]. Others have implemented similar techniques using a compiler framework, resulting in a hybrid static/dynamic analysis framework [20]. There are also analysis tools that use shadow memory to check for safety violations, such as uninitialized memory accesses or memory leaks [21]. However, none of these techniques provide an automated transformation that allows for general simulation of alternate precisions and real-number representations.

The closest work to ours that we are aware of is a system for monitoring the results of *approximate computing*, a class of techniques that trade accuracy for speed at the algorithmic level [22]. This work uses shadow value analysis to track errors introduced by these approximation techniques, but it does not provide support for general real-number representations and is limited to approximated computing applications.

V. FUTURE WORK

While the baseline overhead of our analysis is often less than 50X, the overhead is still an obstacle to large-scale analysis. Although we have already completed some profiling and performance optimization, we hope to reduce the overhead further with additional work. In particular, we anticipate that collapsing instrumentation to one insertion site per basic block will result in considerable improvement without affecting the results. We also anticipate that additional static data flow

analysis would reduce the number of times that shadow values need to be re-initialized.

There are many ways to extend our techniques. We have already begun to implement mechanisms allowing the analysis to report and log events during execution, such as the earliest point at which a shadow value diverges from its original value by a specified amount. We are also interested in detecting when comparisons on shadow values would result in control flow divergence from the original program. Additionally, a mode that computes multiple shadow value types simultaneously would reduce overhead over multiple runs and provide additional opportunities for comparison studies.

In the future, we hope to apply our technique to analyze many more codes, possibly comparing the results against similar techniques implemented in other frameworks. We anticipate that the results of such studies may inform the development of runtime systems for varying floating-point precision to maximize performance and power efficiency while maintaining accuracy.

VI. Conclusion

We present the first automated technique for general shadow value analysis of x86 programs, allowing developers to simulate running double-precision floating-point programs with alternative precision levels or real-numbered representations. Further, we have shown the viability of our technique by building a concrete implementation using Intel Pin and testing it using a variety of benchmarks and applications. This framework serves as a foundation for future research into mixed-precision implementations and alternative representations, reducing the barrier to experimentation and providing insights previously unavailable without extensive manual porting efforts.

Acknowledgment

The authors would like to thank Scott Lloyd et al. from Lawrence Livermore National Laboratory for the C implementation of universal numbers.

References

[1] IEEE, "IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008)," IEEE, New York, Tech. Rep., aug 2008.

[2] D. Goldberg, "What Every Computer Scientist Should Know About Floating Point Arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, mar 1991.

[3] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. van der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The International Exascale Software Project Roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011. [Online]. Available: http://hpc.sagepub.com/cgi/doi/10.1177/1094342010391989

[4] A. Nötzli and F. Brown, "LifeJacket: verifying precise floating-point optimizations in LLVM," in *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis - SOAP 2016*. New York, New York, USA: ACM Press, 2016, pp. 24–29.

[5] J. L. Gustafson, *The End of Error: Unum Computing*. Chapman & Hall/CRC, 2015.

[6] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, pp. 13–es, jun 2007.

[7] 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 *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. New York, NY, USA: ACM, 2005, pp. 190–200.

[8] "The NAS Parallel Benchmarks 3.3. http://www.nas.nasa.gov/publications/npb.html. Accessed 10 January 2014." [Online]. Available: http://www.nas.nasa.gov/publications/npb.html

[9] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 Updates and Changes," Livermore, CA, Tech. Rep., 2013. [Online]. Available: http://codesign.llnl.gov/lulesh

[10] J. H. Wilkinson, "Error Analysis of Floating-point Computation," *Numerische Mathematik*, vol. 2, no. 1, pp. 319–340, dec 1960.

[11] N. J. Higham, *Accuracy and Stability of Numerical Algorithms, Second Edition*. SIAM Philadelphia, 2002.

[12] E. Goubault, "Static Analyses of the Precision of Floating-Point Operations," *Static Analysis*, pp. 234–259, 2001.

[13] M. Martel, "Program Transformation for Numerical Precision," in *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'09)*. New York, NY, USA: ACM Press, jan 2009, pp. 101–110.

[14] W. Lee, R. Sharma, and A. Aiken, "Verifying bit-manipulations of floating-point," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016*, 2016, pp. 70–84.

[15] D. Quinlan, "ROSE: Compiler Support for Object-Oriented Frameworks," *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[16] 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)*, mar 2004.

[17] M. O. Lam, J. K. Hollingsworth, and G. Stewart, "Dynamic Floating-Point Cancellation Detection," *Parallel Computing*, vol. 39, no. 3, pp. 146–155, mar 2013.

[18] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically Adapting Programs for Mixed-Precision Floating-Point Computation," in *Proceedings of the 27th International ACM Conference on Supercomputing (ICS '13)*. New York, New York, USA: ACM Press, jun 2013, p. 369.

[19] M. O. Lam and J. K. Hollingsworth, "Fine-Grained Floating-Point Precision Analysis," *International Journal of High Performance Computing Applications*, p. 1094342016652462, jun 2016.

[20] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning Assistant for Floating-Point Precision," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on (SC'13)*. New York, New York, USA: ACM Press, nov 2013, pp. 1–12.

[21] N. Nethercote and J. Seward, "How to Shadow Every Byte of Memory Used by a Program," *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE 2007)*, p. 65, 2007.

[22] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman, "Monitoring and Debugging the Quality of Results in Approximate Programs," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '15*, vol. 50, no. 4. New York, New York, USA: ACM Press, 2015, pp. 399–411.