A Floating-point Validation Suite for High-performance Shared and Distributed Memory Computing Systems

S. K. Ghoshal
Supercomputer Education and Research Centre
Indian Institute of Science,
Bangalore 560 012 India.

July 11, 1997

Abstract

Methodology to systematically identify and isolate bugs in floating point implementation in high-performance multiple CPU computing systems is formulated. A validation suite is written and tested. Results show improper implementation. Proper implementation guidelines are suggested and prototyped.
Keywords: IEEE754, Validation, Suite, NaN

1 Introduction

High-performance computers often sacrifice correctness in order to achieve throughput, particularly when it comes to floating point operations. "It is the TFLOPS that count, even if a few of the trillion operations are meaningless thereby rendering the end result absurd" - seems to be the attitude of most vendors of computing equipment that deal with floating point operations. This has been observed by the author [1], as well as others [2]. The principles behind correct floating point arithmetic is universally understood and accepted [3]. Floating point Units, that are implemented in hardware at the chip level in most modern microprocessors do a correct and well-meaning implementation [4, 5] of the universally accepted IEEE 754 [6] standards. That is because hardware is much more carefully engineered than system and application software and it is not possible to sell faulty hardware and get away with it easily [7, 9, 8]. However in system software, it is possible to support floating point operations very badly and succeed. A systematic apathy towards the correct implementation of floating point standards at all levels of system software implementation of high-performance computers has resulted in alarming situations where such computers regularly produce completely absurd results without any kind of error or warning messages. And there exist many people (some of whom I know) who accept these results without suspicion. While it is nobody's case to condone such lenient attitude on behalf of the users, what we, the computer scientists and numerical analysts can do is to stem the rot and force the vendors to produce computing environments that implement correct floating point arithmetic. We, at SERC, have taken the following approach:

- Make people aware of these bugs by speaking out at classrooms, conferences, etc., writing literature explaining floating point arithmetic [10] and high-performance FPUs and making it freely available on the Internet [11].
- Write a portable floating point validation suite. Test all the new high-performance machines that get installed. Post the result of the test-suite on the web. Identify the bugs and classify them. Post some recommendation for work-arounds for these bugs on the web. Distribute the source code of the suite freely. In addition to the approach of [12] (which just confines to Fortran 90 and IEEE 754), we discuss a lot of hardware and system software related issues in our literature which we post on the web.
- Discuss the bugs with the representative of the vendor. Initiate steps to rectify them. Maintain the error-indicating versions of suite in such a way that the errors are reproducible quickly at a later point of time. Develop our own operating system micro-kernel and runtime library for floating point operations that implement the IEEE 754 and its error-handling predicates correctly, completely and sensibly. This is for convincing the vendor that the problem is indeed solvable. This keeps

our research interests properly looked after as well. Keep the source code of the RTL on the Web.

This paper is about the second aspect of dealing with the issue. We have developed a floating-point validation suite that works for single-CPU workstations and a number of parallel and distributed high-performance computers. It works both when the underlying microprocessor supports the little-endian format and when it supports the big-endian format. It tests the implementation of IEEE 754 in single precision, double precision and extended precision formats. It tests message passing primitives dealing with floating point variables to check if they transfer NaNs, Infinities, De-normals etc. without any warning/error. This feature is badly required because floating point errors are particularly difficult to track down in distributed memory architectures, as

- the floating point values go from one virtual address space to another disjoint virtual address space very easily and it is difficult to locate the original bug that initiated the syndrome,
- the codes in these machine are often not easily serializable as they depend strongly on the multicomputer configuration on which they are intended to be run, and
- on serialization, many floating point anomalies are not reproducible because they depend on the nominal (normalized) values of the variables in question and such values in turn are dependent on the task-partitioning scheme.

Thus errors must be detected in the actual parallel computing configuration, as topologically closely as the point where it started going absurd. Therefore, our suite does not attempt to serialize and then validate floating point operations. It tracks down the implementation anomalies "in place", as they actually occur in a given multiprocessor/ multicomputer configuration. It understands architectures of parallel and distributed computers. This feature makes it unique among floating point validation suites that exist (*e.g.* the one from NAG [13]) as of now. The rest of the paper is organized as follows. In section 2, we discuss the organization of the validation suite. We tested a number of single-CPU, parallel and distributed computers using this suite. The results that might interest the high-performance computing community are presented in section 3. We conclude in section 4.

2 The suite

The suite tests if the programming environment handles what had been described in the subsections that follow.

2.1 Abnormal IEEE 754 Values

The suite checks for the following:

Quiet NaNs Are they being produced when absurd operations are attempted? Does the system produce a quiet NaN, when quiet NaNs are used as the input operand of a typical floating point operation? In a parallel computer, is the (`nodeid, pid`) pair being embedded/ pointed to by the non-zero bit-field of the QNaN?

Signaling NaNs Do they make the FPU/CPU trap? Is there any way the user can make it trap? Is there any way that the user can supply the trap-handler for SNaNs?

Denormals When any variable goes denormal, does the FPU trap? Can it be made to trap? What is the conversion algorithm from the FPUs internal format to IEEE 754 for denormals? What information is destroyed by the conversion algorithm?

Infinities Are infinities produced correctly? When type-casting is done from one precision to another, do infinities get preserved?

Inexacts What is the way to handle them? Does the presence of one inexact set of a syndrome of a chain of inexacts, thereby rapidly losing precision?

Using this part of the suite is mabpower-intensive and involves cpmprehensive documentation activities too.

2.2 Machine Constants

The suite also computes:

Machine-epsilon The minimum number that makes a significant difference in floating-point computation.

Dwarf The minimum non-zero representable number that can be stored in a IEEE 754 floating point variable.

These are dependent rather on the floating-point format than on the programming language used. We recognize that and the suite allows the user to examine the bit-fields of these two important constants. This also

is the reason, we did not write the suite in Fortran-90 or attempt to portray the properties of the machine constants to be associated with any programming language. Our observation has been that compiler implpementors are equally apathetic to floating point issues. Thus there exists no ideal programming language in which it is the most appropriate to code the suite. We used C as there is a C compiler on all machines and because we can set/test bit patterns in C.

2.3 Rounding Modes and Underflow

The suite also decides the rounding modes that are usable on the FPU, and whether there is any provision for the user to set/test that mode. The suite can decide whether the underflow is gradual or abrupt and whether the user can set/test that feature.

2.4 Parallel Computing Aspects

On Distributed memory machines, the suite tests for the message-passing primitives unsuspectingly passing on any of the abnormal bit-patterns that are listed in section 2.1. On shared-memory machines, there should be a mechanism to trap these values from crossing the virtual address boundaries of processes/tasks, even if the abnormal IEEE 754 floating point value remains in the same physical storage location. I could not, until now, figure out a good way of doing that. Therefore, I could not design such a test in the suite. I just tested for abnormals being able to cross task virtual address space boundaries with gay abandon on shared memory machines.

3 Results

3.1 Coding

The suite was coded in Portable C. As 64-bit integers have to be declared in different ways to compile correctly on C compilers hosted by different high-performance computers, certain `#defines` need to be altered to port the suite. On message-passing machines, the suite uses and links with the message-passing library used for application programming. For example, on the IBM SP2, it uses IBM's MPI [14]. The suite has a large number of routines to set and test bit-patterns for different floating point data types, both normal and abnormal. See, for example, the figures 1, 2 and 3 for the representations of infinities, Signaling NaNs and quiet NaNs, respectively. All optimization switches are turned off when compiling the validation suite code.

3.2 Architecture

We describe three high-performance architectures for which tested the suite.

DEC 8400/300 Turbo Laser It has eight alpha CPUs each at 300 MHz, sharing 2GB of main memory.

IBM SP2 It has 24 distributed memory computing nodes, interconnected by IBM's proprietary High-Performance Switch (HPS). Out of the 24 nodes,

- 8 are 590s, each with 66.67 MHz RS6000/590 CPU, with 512 MB memory
- 16 are 591s, each with 77 MHz RS6000/591 CPU, with 256 MB memory.

The HPS has a link speed of 35 MBytes/sec for point-to-point communication. It has a latency of 40 μseconds.

SGI Power Challenge It has six MIPS R8000 CPU/ R80101 FPU pairs, four with 75 MHz clock speed, and two with 90 MHz clock speed, sharing 1 GB of main memory

3.3 Validation

Table 1 shows some of the important results of the validation runs. There exist implementation deficiencies in all the systems, despite the hardware being capable of supporting a proper implementation.

3.4 Prototyping

In keeping with the third item in the list of steps to tackle improper floating point arithmetic implementation as we mentioned in the introduction as well as to develop our own ways of researching in these issues, we developed a runtime-library/ micro-kernel pair for our own distributed memory architecture built from 16 IBM PC motherboards, each with an Intel 80386/387 and 2MB main memory [15]. As our parallel computer is not a high-performance machine I did not list it in Table 1. However, it handles all the abnormal patterns properly(*i.e*, the way I think they should be handled). It receives all the traps/ signals from the 80387s, handles them by warning the user and/ or terminating the tasks. Abnormal values cannot go from one VA space to another, without causing a trap in the interprocess/ interprocessor communication interface. The

rounding modes can be set by the user. De-normals are reported and handled as per the user's directives. Optionally, the user can write and install her own trap handlers. The RTL initializes all un-initialized floating point variables to SNaNs and the micro-kernel informs the user, whenever such an operand enters the 80387 at any node. The QNaNs keep a record of the `(nodeid, pid)` pair where the NaN was first produced. See [20] for details. Anybody can implement such a micro-kernel, once she has Unix source code of some type [16, 17, 18], reads [4] and has some experience in protected mode programming of 80X86 microprocessors combined with Unix implementation [19].

4 Conclusion

It is time we stop going the teraflop way just for the TFLOPS' sake, and pay a little bit of long-due attention on correctness of floating point operations. It may mean a slight reduction in the execution speed and having to undertake a significant re-design effort of system software. It also will mean having to design, develop and run validation suites appropriate to the many different levels of system software in modern high-performance computers. But the result is certainly worth the effort. In fact, any development in this field is badly awaited and most welcome. If I, coding alone can do it, the big names in high-performance computing can certainly do it too. You may or may not agree with my way of solving the problem or even my way of looking at the problem, but surely you will agree that there is a problem. It is time, therefore, that we all get our act together and make floating point arithmetic correctly implemented on high-performance computers.

Acknowledgements

I thank Dr. Siddhartha Shankar Ghosh. He helped me program the SP2. He also carefully documented certain problems faced by users of the High-performance Computing Systems at SERC, IISc and discussed them with me. That helped me a lot about deciding what needs to be done.

References

[1] S. K. Ghoshal, "A Floating point Validation Suite for the DEC 8400", *Proceedings of the Workshop on recent trends in Parallel and Distributed Computing*, December 23-24, 1996, Department of Computer Science, Pondicherry University, India. Get it by anonymous ftp from `144.16.79.50` as `pub/papers/ghoshal/floating.point.turbolaser.ps`.

[2] David Goldberg, "Computer Arithmetic", Appendix A of Patterson and Hennessy, *Computer Architecture - A Quantitative Approach*, Morgan Kaufmann, 1990.

[3] David Goldberg, "What Every Computer Scientist should know about Floating Point Arithmetic", *ACM Computing Surveys*, pp. 5-48, March 1991.

[4] 80387 Programmer's Reference Manual, Intel Corporation, 1987, Order Number 231917-001, Literature Distribution, Mail Stop SC6-59, 3065 Bowers Avenue, Santa Clara, CA 95051.

[5] Troy, N. Hicks, Richard Fry and Paul E. Harvey, "POWER2 Floating-Point Unit: Architecture and Implementation", `http://bunsen.hrz.uni-marburg.de/hrz/sp/workshop/html/ibmhwsw/fpu.html`

[6] IEEE standard 754-1985 for Binary Floating Point Arithmetic, IEEE. Reprinted in SIGPLAN 22, 2, 9-25.

[7] Floating Point Flaw in the Pentium(tm) Processor `http://www.elc.ees.saitama-u.ac.jp/ kazuhito/intel/fpushort.html`

[8] R. M. Smith, " How Dr. Nicely's Pentium FDIV bug report reached the Internet " Contact Richard M. Smith: `rms@pharlap.com`

[9] H. P. Sarangapani, and M. L. Barton, "Statistical Analysis of Floating Point Flaw in the Pentium Processor", *White Paper by Intel*, Can be had by browsing `http://www.intel.com`

[10] S. K. Ghoshal, "The Dwarf", *Resonance*, page 75, Vol. 1, No. 7, July 1996.

[11] S. K. Ghoshal, "Understanding the IEEE 754 floating point number system", *Lecture notes in Numerical Algorithms for High-performance Computing*, E0201N, Indian Institute of Science, Get it by anonymous ftp from `144.16.79.50` as `pub/papers/ghoshal/ieee754.introduction.ps`.

[12] The Perils of Floating Point, `http://www.lahey.com/float.htm`

[13] The floating point validation suite from the Numerical Algorithms Group (NAG), Inc., 1400 Opus Place, Suite 200, Downers' Grove, IL 60515. Browse `http://www.nag.com/` to order it/ know more about it.

[14] IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference, Version 2, Release 1.

[15] Nanda Kishore, D. and Ghoshal, S. K., "Design, programming environment and applications of a simple low-cost message passing multicomputer", *Journal of Indian Institute of Science*, **76**, May-June 1996, 337-361.

[16] Linux Source code from `http://www.linux.org/`

[17] Linux Source code from `http://www.redhat.com/`

[18] Linux Source code from `http://sunsite.unc.edu/`

[19] S. K. Ghoshal, "Protected Mode Programming and Architectural Support for Memory Management in Intel 80386 and 80486 Microprocessors", Get it by anonymous ftp from `144.16.79.50` as `pub/papers/ghoshal/mmu.80X86.paging.segmentation.ps`.

[20] S. K. Ghoshal, "An Object-oriented Microkernel for Correct Parallel Numerical Linear Algebraic Computation in Distributed Memory Architectures", *Submitted to ADCOMP-97*, Get it by anonymous ftp from `144.16.79.50` as `pub/papers/ghoshal/micro.kernel.ieee754.ps`.

Aspect	Machine		
	IBM SP2	DEC 8400/300	SGI Pow. Ch.
SNaN Trap?	NO	NO	NO
SNaN Detect?	YES	NO	YES
QNaN Detect?	YES	NO	YES
Abnormals Cross VA limit?	YES	YES	YES
Underflow	Gradual	Abrupt	Gradual
De-normal Detect?	YES	NO	NO
Hardware has IEEE754 Compatibility and Capability	YES	YES	YES

Table 1: Summary of Validation Runs

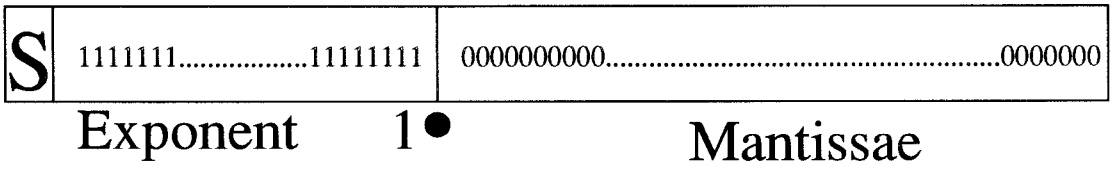

Figure 1: Infinities look like this

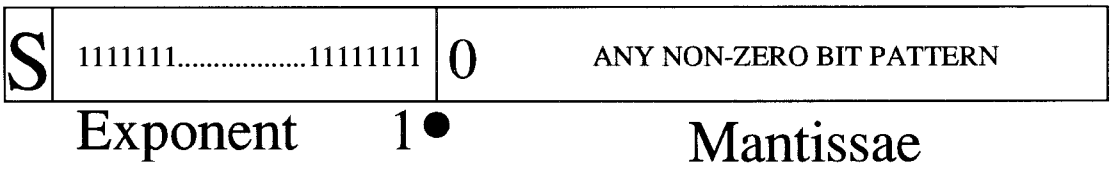

Figure 2: Signaling NaNs look like this

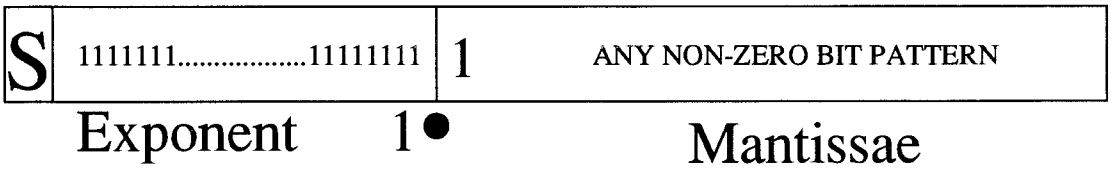

Figure 3: Quiet NaNs look like this