gafferongames.com

Floating Point Determinism

12 min read • original

Introduction

Lately I've been doing some research into networking game physics simulations via deterministic lockstep methods.

The basic idea is that instead of synchronizing the state of physics objects directly by sending the positions, orientations, velocities etc. over the network, one could synchronize the simulation *implicitly* by sending just the player inputs.

This is a very attractive synchronization strategy because the amount of network traffic depends on the size of the player inputs instead of the amount of physics state in the world. In fact, this strategy has been used for many years in RTS games for precisely this reason; with thousands and thousands of units on the map, they simply have too much state to send over the network.

Perhaps you have a complex physics simulation with lots of rigid body state, or a cloth or soft body simulation which needs to stay perfectly in sync across two machines because it is gameplay affecting, but you cannot afford to send all the state. It is clear that the only possible solution in this situation is to attempt a deterministic networking strategy.

But we run into a problem. Physics simulations use floating point calculations, and for one reason or another it is considered very difficult to get exactly the same result from floating point calculations on two different machines. People even report different results on the same machine from run to run, and between debug and release builds. Other folks say that AMDs give different results to Intel machines, and that SSE results are different from x87. What exactly is going on? Are floating point calculations deterministic or not?

Unfortunately, the answer is not a simple “yes” or “no” but a resoundingly limp “maybe?”

Here is what I have discovered so far:

1. If your physics simulation is itself deterministic, with a bit of work you should be able to get it to play back a replay of recorded inputs on the same machine and get the same result.

2. It is possible to get deterministic results for floating calculations across multiple computers provided you use an executable built with the same compiler, run on machines with the same architecture, and perform some platform-specific tricks.

3. It is incredibly naive to write arbitrary floating point code in C or C++ and expect it to give exactly the same result across different compilers or architectures.

4. However with a good deal of work you may be able to coax exactly the same floating point results out of different compilers or different machine architectures by using your compilers “strict” IEEE 754 compliant mode and restricting the set of floating point operations you use. This typically results in significantly lower floating point performance.

If you would like to debate these points or add your own nuance, please write a comment! I consider this question by no means settled and am very interested in other peoples experiences with deterministic floating point simulations and exactly reproducible floating point calculations. Please contact me especially if you have managed to get binary exact results across different architectures and compilers in real world situations.

Here are the resources that I have discovered in my search so far...

> The technology we license to various customers is based on determinism of floating point (in 64-bit mode, even) and has worked that way since the year 2000.
>
> As long as you stick to a single compiler, and a single CPU instruction set, it is possible to make floating point fully deterministic. The specifics vary by platform (i e, different between x86, x64 and PPC).
>
> You have to make sure that the internal precision is set to 64 bits (not 80, because only Intel implements that), and that the rounding mode is consistent. Furthermore, you have to check this after calls to external DLLs, because many DLLs (Direct3D, printer drivers, sound libraries, etc) will change the precision or rounding mode without setting it back.
>
> The ISA is IEEE compliant. If your x87 implementation isn't IEEE, it's not x87.
>
> Also, you can't use SSE or SSE2 for floating point, because it's too under-specified to be deterministic.

Jon Watte, GameDev.net forums
http://www.gamedev.net/community/forums/topic.asp?topic_id=499435

> I work at Gas Powered Games and i can tell you first hand that floating point math is deterministic. You just need the same instruction set and compiler and of course the user's processor adheres to the

IEEE754 standard, which includes all of our PC and 360 customers. The engine that runs DemiGod, Supreme Commander 1 and 2 rely upon the IEEE754 standard. Not to mention probably all other RTS peer to peer games in the market. As soon as you have a peer to peer network game where each client broadcasts what command they are doing on what 'tick' number and rely on the client computer to figure out the simulation/physical details your going to rely on the determinism of the floating point processor.

At app startup time we call:

```
_controlfp(_PC_24, _MCW_PC)
_controlfp(_RC_NEAR, _MCW_RC)
```

Also, every tick we assert that these fpu settings are still set:

```
gpAssert( (_controlfp(0, 0) & _MCW_PC) == _PC_24 );
gpAssert( (_controlfp(0, 0) & _MCW_RC) == _RC_NEAR );
```

There are some MS API functions that can change the fpu model on you so you need to manually enforce the fpu mode after those calls to ensure the fpu stays the same across machines. The assert is there to catch if anyone has buggered the fpu mode.

FYI We have the compiler floating point model set to Fast /fp:fast (but its not a requirement)

We have never had a problem with the IEEE standard across any PC cpu AMD and Intel with this approach. None of our SupCom or Demigod customers have had problems with their machines either, and we are talking over 1 million customers here (supcom1 + expansion pack). We would have heard if there was a problem with the fpu not having the same results as replays or multiplayer mode wouldn't work at all.

We did however have problems when using some physics APIs because their code did not have determinism or reproducibility in mind. For example some physics APIS have solvers that take X

> number of iterations when solving where X can be lower with faster CPUs.

Elijah, Gas Powered Games

http://www.box2d.org/forum/viewtopic.php?f=3&t=1800

> If you store replays as controller inputs, they cannot be played back on machines with different CPU architectures, compilers, or optimization settings. In MotoGP, this meant we could not share saved replays between Xbox and PC. It also meant that if we saved a replay from a debug build of the game, it would not work in release builds, or vice versa. This is not always a problem (we never shipped debug builds, after all), but if we ever released a patch, we had to build it using the exact same compiler as the original game. If the compiler had been updated since our original release, and we built a patch using the newer compiler, this could change things around enough that replays saved by the original game would no longer play back correctly.
>
> This is madness! Why don't we make all hardware work the same? Well, we could, if we didn't care about performance. We could say "hey Mr. Hardware Guy, forget about your crazy fused multiply-add instructions and just give us a basic IEEE implementation", and "hey Compiler Dude, please don't bother trying to optimize our code". That way our programs would run consistently slowly everywhere ☺

Shawn Hargreaves, MSDN Blog

http://blogs.msdn.com/shawnhar/archive/2009/03/25/is-floating-point-math-deterministic.aspx

> "Battlezone 2 used a lockstep networking model requiring absolutely identical results on every client, down to the least-significant bit of the mantissa, or the simulations would start to diverge. While this was difficult to achieve, it meant we only needed to send user input across the network; all other game state could be computed locally. During development, we discovered that AMD and Intel processors produced slightly different results for trancendental functions (sin, cos, tan, and their inverses), so we had to wrap them in non-optimized

> function calls to force the compiler to leave them at single-precision. That was enough to make AMD and Intel processors consistent, but it was definitely a learning experience.

Ken Miller, Pandemic Studios

http://www.box2d.org/forum/viewtopic.php?f=4&t=175

> ... In FSW1 when desync is detected in player would be instantly killed by “magic sniper”. ☺ All that stuff was fixed in FSW2. We just ran precise FP and used Havok FPU libs instead SIMD on PC. Also integer modulo is problem too because C++ standard says it’s “implementation defined” (in case when multiple compilers/platforms are used). In general I liked tools for lockstep we developed, finding desyncs in code on FSW2 was trivial.

Branimir Karadžić, Pandemic Studios

http://www.google.com/buzz/100111796601236342885/8hDZ655S6x3/F
Point-Determinism-Gaffer-on-Games

> I know three main sources of floating point inconsistency pain:
>
> Algebraic compiler optimizations
> “Complex” instructions like multiply-accumulate or sine
> x86-specific pain not available on any other platform; not that ~100% of non-embedded devices is a small market share for a pain.
>
> The good news is that most pain comes from item 3 which can be more or less solved automatically. For the purpose of decision making (”should we invest energy into FP consistency or is it futile?”), I’d say that it’s not futile and if you can cite actual benefits you’d get from consistency, then it’s worth the (continuous) effort.
>
> Summary: use SSE2 or SSE, and if you can’t, configure the FP CSR to use 64b intermediates and avoid 32b floats. Even the latter solution works passably in practice, as long as everybody is aware of it.

Yossi Kreinin, Consistency: how to defeat the purpose of IEEE floating point

http://www.yosefk.com/blog/consistency-how-to-defeat-the-purpose-of-ieee-floating-point.html

> The short answer is that FP calculations are entirely deterministic, as per the IEEE Floating Point Standard, but that doesn't mean they're entirely reproducible across machines, compilers, OS's, etc.
>
> The long answer to these questions and more can be found in what is probably the best reference on floating point, David Goldberg's What Every Computer Scientist Should Know About Floating Point Arithmetic. Skip to the section on the IEEE standard for the key details.
>
> Finally, if you are doing the same sequence of floating point calculations on the same initial inputs, then things should be replayable exactly just fine. The exact sequence can change depending on your compiler/os/standard library, so you might get some small errors this way.
>
> Where you usually run into problems in floating point is if you have a numerically unstable method and you start with FP inputs that are approximately the same but not quite. If your method's stable, you should be able to guarantee reproducibility within some tolerance. If you want more detail than this, then take a look at Goldberg's FP article linked above or pick up an intro text on numerical analysis.

Todd Gamblin, Stack Overflow

http://stackoverflow.com/questions/968435/what-could-cause-a-deterministic-process-to-generate-floating-point-errors

> The C++ standard does not specify a binary representation for the floating-point types float, double and long double. Although not required by the standard, the implementation of floating point arithmetic used by most C++ compilers conforms to a standard, IEEE 754-1985, at least for types float and double. This is directly related to the fact that the floating point units of modern CPUs also support this

> standard. The IEEE 754 standard specifies the binary format for floating point numbers, as well as the semantics for floating point operations. Nevertheless, the degree to which the various compilers implement all the features of IEEE 754 varies. This creates various pitfalls for anyone writing portable floating-point code in C++.

Günter Obiltschnig, Cross-Platform Issues with Floating-Point arithmetics in C++
http://www.appinf.com/download/FPIssues.pdf

> Floating-point computations are strongly dependent on the FPU hardware implementation, the compiler and its optimizations, and the system mathematical library (libm). Experiments are usually reproducible only on the same machine with the same system library and the same compiler using the same options.

STREFLOP Library
http://nicolas.brodu.numerimoire.net/en/programmation/streflop/inde

> Floating Point (FP) Programming Objectives:
>
> • **Accuracy** – Produce results that are “close” to the correct value
>
> • **Reproducibility** – Produce consistent results from one run to the next. From one set of build options to another. From one compiler to another. From one platform to another.
>
> • **Performance** – Produce the most efficient code possible.
>
> These options usually conflict! Judicious use of compiler options lets you control the tradeoffs.

Intel C++ Compiler: Floating Point Consistency
http://www.nccs.nasa.gov/images/FloatingPoint%5Fconsistency.pdf.

> If strict reproducibility and consistency are important do not change the floating point environment without also using either fp-model strict (Linux or Mac OS*) or /fp:strict (Windows*) option or pragma fenv_access.

Intel C++ Compiler Manual

http://cache-www.intel.com/cd/00/00/34/76/347605_347605.pdf

> Under the fp:strict mode, the compiler never performs any optimizations that perturb the accuracy of floating-point computations. The compiler will always round correctly at assignments, typecasts and function calls, and intermediate rounding will be consistently performed at the same precision as the FPU registers. Floating-point exception semantics and FPU environment sensitivity are enabled by default. Certain optimizations, such as contractions, are disabled because the compiler cannot guarantee correctness in every case.

Microsoft Visual C++ Floating-Point Optimization

http://msdn.microsoft.com/en-us/library/aa289157(VS.71).aspx#floapoint_topic4

> Please note that the results of floating point calculations will likely not be exactly the same between PowerPC and Intel, because the PowerPC scalar and vector FPU cores are designed around a fused multiply add operation. The Intel chips have separate multiplier and adder, meaning that those operations must be done separately. This means that for some steps in a calculation, the Intel CPU may incur an extra rounding step, which may introduce 1/2 ulp errors at the multiplication stage in the calculation.

Apple Developer Support

http://developer.apple.com/hardwaredrivers/ve/sse.html

> For all of the instructions that are IEEE operations (*,+,-,/,sqrt, compares, regardless of whether they are SSE or x87), they will produce the same results across platforms with the same control settings (same precision control and rounding modes, flush to zero, etc.) and inputs. This is true for both 32-bit and 64-bit processors... On the x87 side, the transcendental instructions like, fsin, fcos, etc. could

> produce slightly different answers across implementations. They are specified with a relative error that is guaranteed, but not bit-for-bit accuracy.

Intel Software Network Support
http://software.intel.com/en-us/forums/showthread.php?t=48339

> I'm concerned about the possible differences between hardware implementations of IEEE-754. I already know about the problem of programming languages introducing subtle differences between what is written in the source code and what is actually executed at the assembly level. [Mon08] Now, I'm interested in differences between, say, Intel/SSE and PowerPC at the level of individual instructions.

D. Monniaux on IEEE 754 mailing list
http://grouper.ieee.org/groups/754/email/msg03864.html

> One must ... avoid the non-754 instructions that are becoming more prevalent for inverse and inverse sqrt that don't round correctly or even consistently from one implementation to another, as well as the x87 transcendental operations which are necessarily implemented differently by AMD and Intel.

David Hough on 754 IEEE mailing list
http://grouper.ieee.org/groups/754/email/msg03867.html

> Yes, getting reproducible results IS possible. But you CAN'T do it without defining a programming methodology intended to deliver that property. And that has FAR more drastic consequences than any of its proponents admit – in particular, it is effectively incompatible with most forms of parallelism.

Nick Maclaren on 754 IEEE mailing list
http://grouper.ieee.org/groups/754/email/msg03872.html

> If we are talking practicabilities, then things are very different, and expecting repeatable results in real programs is crying for the moon. But we have been there before, and let's not go there again.

Nick Maclaren on 754 IEEE mailing list

http://grouper.ieee.org/groups/754/email/msg03862.html

> The IEEE 754-1985 allowed many variations in implementations (such as the encoding of some values and the detection of certain exceptions). IEEE 754-2008 has tightened up many of these, but a few variations still remain (especially for binary formats). The reproducibility clause recommends that language standards should provide a means to write reproducible programs (i.e., programs that will produce the same result in all implementations of a language), and describes what needs to be done to achieve reproducible results.

Wikipedia Page on IEEE 754-2008 standard

http://en.wikipedia.org/wiki/IEEE_754-2008#Reproducibility

> If one wants semantics almost exactly faithful to strict IEEE-754 single or double precision computations in round-to-nearest mode, including with respect to overflow and underflow conditions, one can use, at the same time, limitation of precision and options and programming style that force operands to be systematically written to memory between floating-point operations. This incurs some performance loss; furthermore, there will still be slight discrepancy due to double rounding on underflow.
>
> A simpler solution for current personal computers is simply to force the compiler to use the SSE unit for computations on IEEE-754 types; however, most embedded systems using IA32 microprocessors or microcontrollers do not use processors equipped with this unit.

David Monniaux, The pitfalls of verifying floating-point computations

http://hal.archives-ouvertes.fr/docs/00/28/14/29/PDF/floating-point-article.pdf

> 6. REPRODUCIBILITY

> Even under the 1985 version of IEEE-754, if two implementations of the standard executed an operation on the same data, under the same rounding mode and default exception handling, the result of the operation would be identical. The new standard tries to go further to describe when a program will produce identical floating point results on different implementations. The operations described in the standard are all reproducible operations.
>
> The recommended operations, such as library functions or reduction operators are not reproducible, because they are not required in all implementations. Likewise dependence on the underflow and inexact flags is not reproducible because two different methods of treating underflow are allowed to preserve conformance between IEEE-754(1985) and IEEE-754(2008). The rounding modes are reproducible attributes. Optional attributes are not reproducible.
>
> The use of value-changing optimizations is to be avoided for reproducibility. This includes use of the associative and disributative laws, and automatic generation of fused multiply-add operations when the programmer did not explicitly use that operator.

Peter Markstein, The New IEEE Standard for Floating Point Arithmetic

http://drops.dagstuhl.de/opus/volltexte/2008/1448/pdf/08021.Markste

> Unfortunately, the IEEE standard does not guarantee that the same program will deliver identical results on all conforming systems. Most programs will actually produce different results on different systems for a variety of reasons. For one, most programs involve the conversion of numbers between decimal and binary formats, and the IEEE standard does not completely specify the accuracy with which such conversions must be performed. For another, many programs use elementary functions supplied by a system library, and the standard doesn't specify these functions at all. Of course, most programmers know that these features lie beyond the scope of the IEEE standard.

> Many programmers may not realize that even a program that uses only the numeric formats and operations prescribed by the IEEE standard can compute different results on different systems. In fact, the authors of the standard intended to allow different implementations to obtain different results. Their intent is evident in the definition of the term destination in the IEEE 754 standard: "A destination may be either explicitly designated by the user or implicitly supplied by the system (for example, intermediate results in subexpressions or arguments for procedures). Some languages place the results of intermediate calculations in destinations beyond the user's control. Nonetheless, this standard defines the result of an operation in terms of that destination's format and the operands' values." (IEEE 754-1985, p. 7) In other words, the IEEE standard requires that each result be rounded correctly to the precision of the destination into which it will be placed, but the standard does not require that the precision of that destination be determined by a user's program. Thus, different systems may deliver their results to destinations with different precisions, causing the same program to produce different results (sometimes dramatically so), even though those systems all conform to the standard.

Differences Among IEEE 754 Implementations
http://docs.sun.com/source/806-3568/ncg_goldberg.html#3098

If you enjoyed this article please consider making a small donation.
Donations encourage me to write more articles!

Original URL:

http://gafferongames.com/networking-for-game-programmers/floating-point-determinism/