Spying on the Floating Point Behavior of Existing, Unmodified Scientific Applications

Peter Dinda
Northwestern University

Alex Bernat
Northwestern University

Conor Hetland
Northwestern University

Abstract

Scientific (and other) applications are critically dependent on calculations done using IEEE floating point arithmetic. A number of concerns have been raised about correctness in such applications given the numerous gotchas the IEEE standard presents for developers, as well as the complexity of its implementation at the hardware and compiler levels. The standard and its implementations do provide mechanisms for analyzing floating point arithmetic as it executes, making it possible to find and track problematic operations. However, this capability is seldom used in practice. In response, we have developed FPSpy, a tool that provides this capability when operating underneath existing, unmodified x64 application binaries on Linux, including those using thread- and process-level parallelism. FPSpy can observe application behavior without any cooperation from the application or developer, and can potentially be deployed as part of a job launch process. We present the design, implementation, and performance evaluation of FPSpy. FPSpy operates conservatively, getting out of the way if the application itself begins to use any of the OS or hardware features that FPSpy depends on. Its overhead can be throttled, allowing a tradeoff between which and how many unusual events are to be captured, and the slowdown incurred by the application, with the low point providing virtually zero slowdown. We evaluated FPSpy by using it to methodically study seven widely-used applications/frameworks from a range of domains (five of which are in the NSF XSEDE top-20), as well as the NAS and PARSEC benchmark suites. All told, these comprise about 7.5 million lines of source code in a wide range of languages, and parallelism models (including OpenMP and MPI). FPSpy was able to produce trace information for all of them. The traces show that problematic floating point events occur in both the applications and the benchmarks. Analysis of the rounding behavior captured in our traces also suggests the feasibility of an approach to adding adaptive precision underneath existing, unmodified binaries.

CCS Concepts

• **Software and its engineering → Correctness**; **Software reliability**; **Operational analysis**; • **Mathematics of computing → Numerical analysis**; **Arbitrary-precision arithmetic**.

Keywords

floating point arithmetic, software development, IEEE 754

ACM Reference Format:
Peter Dinda, Alex Bernat, and Conor Hetland. 2020. Spying on the Floating Point Behavior of Existing, Unmodified Scientific Applications. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '20), June 23–26, 2020, Stockholm, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3369583.3392673

This project was supported by the United States National Science Foundation via grants CCF-1533560 and CNS-1763743. Jin Han and John Albers also contributed.

HPDC '20, June 23–26, 2020, Stockholm, Sweden

ACM ISBN 978-1-4503-7052-3/20/06…$15.00
https://doi.org/10.1145/3369583.3392673

1 Introduction

Virtually all applications in scientific and engineering domains, as well as applications built on machine learning techniques, make extensive use of IEEE 754 floating point arithmetic through its numerous implementations. Floating point has proven to be extremely effective at enabling high performance while providing behavior that is sensible to a knowledgeable developer.

However, despite a superficial similarity, floating point arithmetic is not real number arithmetic, and the intuitive framework one might carry forward from real number arithmetic rarely applies. Furthermore, as hardware and compiler optimizations rapidly evolve, it is challenging even for a knowledgeable developer to keep up. In short, floating point and its implementations present sharp edges for its users, and the edges are getting sharper. Recently, a first-of-its-kind study [15] evaluated ~250 developers to get at their understanding of the floating point standard itself, as well as several aspects of implementations. The results were quite mixed. In a quiz of floating point's "gotcha"s, participants performed similarly to what would be expected by chance. In small quiz of implementation aspects, most participants chose "don't know". Finally, participants were arguably less suspicious than would be ideal of the various events that could occur during execution, events that FPSpy specifically captures. This study suggests developers may be inadvertently introducing problematic floating point into scientific applications.

Equally concerning, recent research has also determined that the increasing variability of floating point implementations at the compiler (including optimization choices) and hardware levels is leading to divergent scientific results. For example, NCAR had significant issues in porting the Community Earth Science Model (CESM) to a new platform [31], enough so that they developed their own analysis toolchain for this specific, very important codebase [30]. Their goal is to minimize developer effort in resolving the issues of future ports. Bentley et al have developed a general purpose tool that creates numerous variations during the build process, runs the variants, finds those that produce varying outputs, and then employs a bisection process to point to the source code that is vulnerable to the variation [8, 42]. Through such work, and others, it is becoming clear that the sharp edges of floating point are indeed catching on scientific programs.

We add a new capability to this mix, namely the ability to spy

on the floating point instructions of an existing, unmodified x64 binary running on Linux. The IEEE standard provides for discovery and tracking of several events that can occur at the level of individual operations, such as an add instruction in the ISA. This functionality is exposed architecturally in x64 systems as part of the SSE* and AVX* ISAs, but it can be a challenge to use. As we note in Section 5.1, the direct use of this capability, even when abstracted, appears to be quite rare. Our tool, FPSpy, uses this capability *seamlessly* "underneath" the application, without application or developer knowledge. Any application can be traced using FPSpy to find which events occur, as well as to pinpoint the specific instructions and other context that cause events. The overhead of FPSpy varies considerably depending on what is observed.

FPSpy operates independently of the language, run-time, libraries, and even parallelism model of the application. Indeed, it works with existing, unmodified application *binaries*, and one of its motivating use-cases is to surreptitiously spy on the stream of preexisting jobs on a production machine. The FPSpy capability can operate without any user or developer cooperation.

We evaluate FPSpy by testing it on a wide range of scientific applications and benchmarks, comprising over 7.5 million lines of code, written in diverse languages and using threading, OpenMP, and MPI. This evaluation also serves to produce input data for a study of these codebases to provide data for the question of to what extent to scientific codebases exhibit problematic floating point behavior. Such behavior does exist.

Our contributions are as follows.

- We present the design, implementation, and evaluation of the publicly available FPSpy tool. As far as we are aware, FPSpy is the first tool to enable the in-situ monitoring of floating point behavior of existing, unmodified application binaries without requiring any user or developer cooperation.
- We apply FPSpy to a range of benchmarks and applications, and in particular to large scale applications/application frameworks of significant scientific note. FPSpy is robust.
- We find a range of possible problematic floating point behavior in the applications and benchmarks.
- We characterize rounding behavior in the applications and benchmarks at the instruction level from the perspective of building a trap-and-emulate-based system to mitigate rounding through enhanced precision.

2 Motivation, use-cases, and requirements

The overall motivation behind FPSpy is to collect detailed data to understand the floating point behavior of production scientific applications with an eye toward improving those applications. We want to bridge between what the user actually does and what the analyst sees. A key requirement, which is significantly different from related work (Section 7), is that no developer or user cooperation can be expected (or needed).

Our motivation suggested three use-cases, which are illustrated in Figure 1. Of these, the most demanding is *spying in production* (Figure 1(a)), which drove many of our design decisions. Here, the goal is to track track floating point behavior within production codes as they are being used to do real science on production machines. The user would submit their job as normal. When the job is

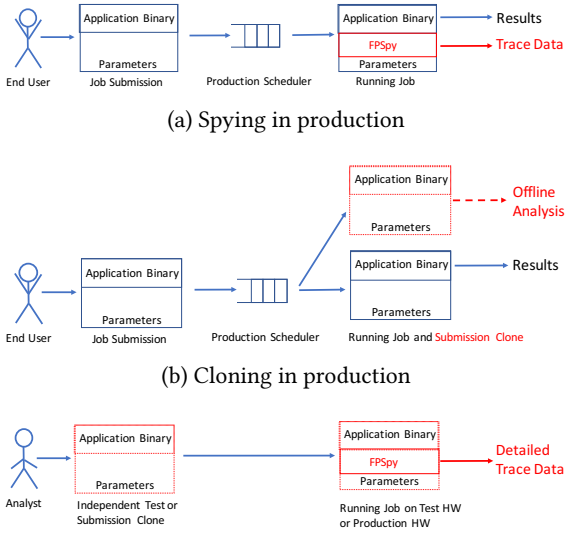

Figure 1: Motivating use-cases for FPSpy.

launched by the scheduler, the launch process would augment all of the job's processes with FPSpy. As the augmented job runs, FPSpy would work in tandem, collecting trace data independently of the application. From the user's perspective, nothing would change, but now each job would provide information for analysts and others interested in studying and improving floating point behavior. Additionally, particularly problematic behavior could be red-flagged.

A second use-case, *cloning in production* (Figure 1(b)) , would capture the job and the scheduler's decisions at the point of job launch, creating a *submission clone*. The submission clone would not be run, but rather used for offline analysis. Again, from the user's perspective, nothing would have changed, and their results would be delivered with no overhead at all. However, the submission clone could be run later using FPSpy.

The final use-case, *spying in the lab* (Figure 1(c)) involves having the analyst run the application on their own. Alternatively, the analyst can use a submission clone (as in the previous use-case) as a starting point, tuning the job parameters to fit the available resources if absolutely necessary. As with spying in production, the application is then run, augmented with FPSpy, to produce trace data. Because the user is out of the loop, we could use FPSpy much more aggressively, resulting in both greater slowdowns and much more detailed trace data.

These use-cases produce the following requirements. To the best of our knowledge, no existing tool meets them.

- FPSpy must operate on real applications, not just benchmarks. No one cares about the floating point behavior of benchmarks other than to what extent it might be representative of real applications.
- FPSpy must operate on existing, unmodified application binaries. Recompilation/binary rewriting is unacceptable. This is driven by the all three use-cases, since the goal is to study the behavior of the applications as it occurs in practice. As we noted in the introduction, it is now well understood that

Variable	Explanation
LD_PRELOAD	Add FPSpy to run
FPE_MODE	Mode of operation (aggregate, individual)
FPE_AGGRESSIVE	Controls for when to "get out of the way"
FPE_DISABLE	Controls for when to "get out of the way"
FPE_EXCEPT_LIST	Filter events to capture
FPE_MAXCOUNT	Disable FPSpy after this many events
FPE_SAMPLE	Downsample the event stream
FPE_POISSON	Use Poisson temporal sampling
FPE_TIMER	Select real-time or instruction time

Figure 2: The interface of FPSpy is environment variables.

compilers and hardware features can produce significant effects on floating point behavior. As a consequence, it is essential to see the exact instructions that will be executed (and, ideally, on the exact machine on which they are executed.)

- FPSpy must be able to support applications that use shared memory (e.g. OpenMP) and distributed memory (e.g. MPI) parallelism. This support should be based on widely available kernel-level functionality, not on specific libraries/run-times.
- FPSpy must not perturb the application in any way other than timing. This severely constrains the ways in which FPSpy can use libraries itself. Much of the FPSpy engine will have to be self-hosting. Additionally, timing perturbation must be limited for the spying in production use-case.[1]
- FPSpy must be able to "get out of the way" the instant the application it is observing begins to use any functionality that FPSpy depends on. It cannot simply abort, but rather must be able to gracefully untangle itself while allowing the application to continue to run. This is an essential requirement for use in the spying in production use-case. For the spying in the lab use case, this requirement can be relaxed by the analyst based on their judgment.
- FPSpy's tradeoff between overhead and information must be adjustable. In the spying in production use case, virtually no overhead can be permitted. On the other hand, in the cloning in production and spying in the lab use-cases, substantial overhead may be acceptable provided fine-grain (instruction-level) information is extracted.

It is important to note that there is a great diversity of scientific applications, libraries, frameworks, and even language implementations, some of which have have form of support for tracking floating point behavior. A core goal of FPSpy is to be independent of these. FPSpy should be applicable to *any* program.

3 System design

FPSpy builds on hardware features that detect exceptional floating point events as a side-effect of normal processing. It also builds on the standard Linux interfaces that allow capture of these events from user level, as well as linker-level mechanisms to allow it to interpose on programs. FPSpy is implemented entirely at the user level and requires no special privileges to use.

[1]Timing perturbation means that some instructions may take longer than normal. Changing timing can reveal the presence of preexisting race conditions in concurrent applications. Such a race condition is a bug in the original application. FPSpy does not have the goal of finding such a bug, but if it exists, it important to note that it has likely already been affecting results, silently, prior to any use of FPSpy.

Condition	Explanation
Inexact	Result is a rounded version of true result
Underflow	Result was a denorm or zero (actual result did not fit)
Overflow	Result was an infinity (actual result did not fit)
Denorm	Operand is a denormalized number (x64-specific)
DivideByZero	Attempt to divide by zero
Invalid	Operand is not a number (NaN)

Figure 3: Events observed by FPSpy are detected by hardware.

3.1 Interface

To support the use-cases of Figure 1, FPSpy has a configuration interface based entirely on environment variables that allows it to be wrapped around any command in any system. Given an application launched via a complex command such as

```
app -i inputs -o outputs -l > app.txt
```

the FPSpy equivalent simply adds environment variables:

```
[FPSPY_VARS] app -i inputs -o outputs -l > app.txt
```

Note that this also allows FPSpy to be used in models where the executable is launched in an indirect manner, such as MPI's `mpirun`. FPSpy internally handles new process and thread creations, which inherit the environment variables, and thus the settings. Figure 2 shows the current set of environment variables used to instantiate and control FPSpy. The only required variables are `LD_PRELOAD`, which adds FPSpy, and `FPE_MODE`, which selects the operation mode.

There are two modes of operation. Aggregate-mode minimizes overhead at the expense of information capture, and gives a single, human-readable trace record for each thread in the program. Individual-mode can capture a trace record for each dynamic floating point instruction and allows a tradeoff between overhead and detail. Individual-mode trace records are in a binary form suitable for being `mmap()`ed into analysis programs for speed. Scripts are provided to turn them into human readable forms, and for analysis.

3.2 Condition codes and exception masking

The IEEE floating point standard defines five condition codes, while x64 adds an additional one. These correspond to the events FPSpy observes, which are described in Figure 3. The condition codes are set as a zero-cost side effect of each floating point operation. In the case of a vector instruction, the condition value generated is an or of the condition values generated by the elementwise operations of the instruction. Unlike the integer condition codes many developers are familiar with on architectures like x64, the floating point condition codes are *sticky*, meaning that once a condition code is set, it remains set until explicitly cleared. This property is used in FPSpy's aggregate-mode.

Each condition code has a corresponding exception mask. When the mask is disabled, the setting of the condition code (even if it is already set) causes a precise exception to be generated before the instruction writes back. Strictly speaking, one operation can result in multiple condition codes being set. A priority encoding determines which exception is delivered. This is outside of the scope of this paper. FPSpy unmasks exceptions when running in individual-mode. It records not only the specific exception delivered, but also the state of all the condition codes.

One subtlety involves NaNs, which are representations of non-numbers. There are two categories of NaNs: QNaNs and SNaNs (also called signaling NaNs). The Invalid condition code captures both, but an Invalid exception is generally only raised if at least one

operand is an SNaN. As a consequence, FPSpy in individual-mode can undercount NaNs.

FPSpy's implementation is currently specific to x64, and it directly uses the x64 `%mxcsr` floating point control/status register, which provides condition codes, exception masks, rounding mode control, and several other control mechanisms for SSE* and AVX* operation, the common floating point model on x64 machines today.

3.3 Interposition

FPSpy is implemented as an `LD_PRELOAD` shared library that uses standard techniques to interpose on a range of functions. When the application is being executed, the dynamic linker of the system loads FPSpy first, and then resolves references within the application against it. FPSpy itself later explicitly resolves references against subsequently loaded libraries, such as `libc`, `libpthread`, and others. The specific set of functions interposed on are shown in Figure 8; they boil down to process and thread management, signal hooking, and floating point environment control.

Process and thread management functions are interposed upon to allow FPSpy to (recursively) follow thread and process forks. FPSpy produces an independent trace for every thread within the process tree rooted at the originally executed application.

Signal hooking and floating point environment control functions are interposed upon because when these are used, it may indicate that FPSpy must "get out of the way". For example, if the application changes the floating point exception mask or clears the floating point exception state, this indicates that it uses functionality that could be perturbed by FPSpy's use of the same functionality. Similarly, in individual-mode operation, FPSpy relies on being able to leverage two to three signals for its own purposes. If the application also attempts to use any of them, FPSpy must disable itself.

Aggression: In individual-mode, FPSpy will, by default, disable itself when certain signals (`SIGTRAP`, `SIGFPE`, and in some configurations, an alarm signal) are used by the application. We have found some applications which use these signals do so only incidentally. It is possible to run FPSpy in "aggressive-mode", in which case it will not step aside when this specific scenario happens. This is particularly useful in a lab environment.

3.4 Initialization and teardown

FPSpy uses the linker's constructor and destructor attributes (typically used to support initialization and teardown of C++ objects with global scope) to hook its own initialization and teardown code into the execution of the application. These occur prior to and after `main()`, respectively. During initialization, FPSpy configures itself in one of its two modes, each of which has a corresponding hardware configuration, and state machine during operation. This constructor-driven initialization is also used to follow process forks. The child process simply inherits the `LD_PRELOAD` and FPSpy configuration environment variables.

On a thread fork (e.g., `clone()` or `pthread_create()`), a thunk is used instead of the application-supplied function. The thunk does an initialization for the thread before invoking the application-supplied function, and a teardown once it finishes. FPSpy's thread initialization and teardown are similar to those of a process. On teardown of either, the corresponding trace file is completed.

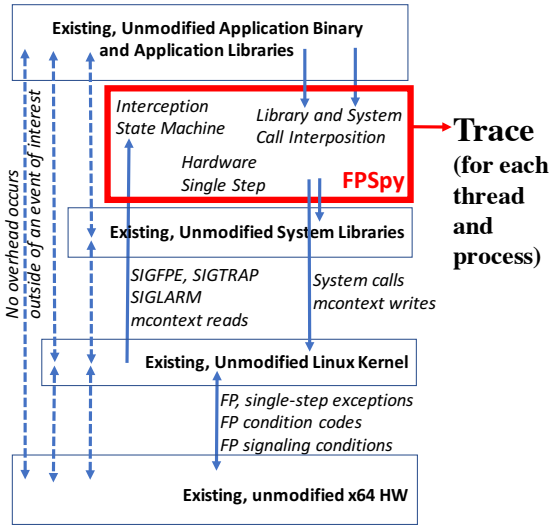

Figure 4: FPSpy's system design is self-contained.

3.5 Aggregate-mode operation

In aggregate-mode, FPSpy has the minimum possible overhead and overlap with the application's own operation, but also produces the least information in its trace.

On initialization for a thread or process, FPSpy clears all condition codes in the floating point control/status register. On teardown, FPSpy records the state of all the condition codes. Recall that the floating point condition codes are "sticky", meaning that once set, they must be *explicitly* cleared. As a consequence, if a particular floating point event occurs at least once in the thread being spied upon, the corresponding condition code will be set when the thread teardown occurs. In other words, in aggregate-mode, FPSpy records the set of events that occurred during the execution of the thread. It does not record the number of such events nor the specific instructions causing them.

It is important to understand just how low the overhead of this mode of operation can be. The condition codes are already being set by the hardware during each floating point instruction. The cost of aggregate-mode is ultimately simply a write of `%mxcsr` at the beginning of a thread's life cycle, and a read at the end of it.

3.6 Individual-mode operation

FPSpy's individual-mode is considerably more complicated because it can capture the context of every single instruction that causes a floating point event during the execution of a thread. In order to trade off between overhead and information gathered, a range of sampling mechanisms are included. Figure 4 illustrates the run-time operation and structure of FPSpy when running in individual-mode. The overall structure also applies for aggregate-mode. Figure 5 illustrates the state machine that is used by FPSpy when monitoring a thread in individual-mode.

On initialization in individual-mode, FPSpy configures the hardware's floating point control/status register such that every monitored event will cause an exception. It also installs a signal handler for the `SIGFPE` (floating point exception) and `SIGTRAP` (single-stepping trap) signals. It then creates a state machine for the thread,

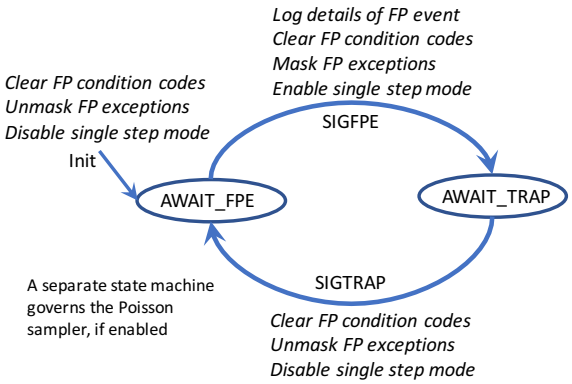

Figure 5: FPSpy's per-thread individual-mode state machine.

sets the initial state to AWAIT_FPE, clears the floating point condition codes, and then lets the thread start.

Consider now what happens if the thread executes a floating point instruction that causes one of the floating point events. The hardware will raise an exception. The kernel will translate that exception into a `SIGFPE` and invoke FPSpy's installed handler for this signal. The handler will use the contextual information passed on invocation to find the monitoring context and state machine for the thread. The handler can then record the specific event that has occurred. It records the timestamp, the instruction pointer, the instruction data, the stack pointer, the kernel-supplied floating point control and status information, and the value of `%mxcsr`.

At this point, we need to allow the instruction to execute as it normally would have in the application, but we need to regain control immediately afterward. To do this, we switch to the AWAIT_TRAP state. We also clear the floating point condition codes, and mask the floating point exceptions. This is done by manipulating `%mxcsr`. Masking the floating point exceptions will allow the instruction to run without faulting again. However, we want only a single instruction to run. To assure this, we set the trap bit in the x64 `%rflags` register. This switches the thread into single-stepping mode—after every instruction a trap exception will occur. Having configured the machine, and our state in this way, we next return from the signal handler, which causes the kernel to restart the faulting instruction.

Immediately after the instruction finishes, the hardware will trigger the trap exception, because we have placed it in single-step mode. The kernel handles the exception by sending the application a `SIGTRAP`, which FPSpy duly handles. The `SIGTRAP` handler finds the corresponding monitoring context and state machine for the thread. It then clears the floating point condition codes, unmasks the floating point exceptions, and deactivates single-stepping mode. It switches back to the AWAIT_FPE state, and is now ready to handle the next faulting floating point instruction. The signal handler then returns to the kernel, which restarts the instruction that caused the `SIGTRAP`, which is the instruction immediately subsequent to the original faulting floating point instruction.

In essence, FPSpy in individual-mode implements a classic trap-and-emulate model for faulting instructions, a common model for virtual machine monitors/hypervisors, but it does so entirely at user level. "Emulation" here consists of simply running the faulting instruction after we have recorded the circumstances of its fault.

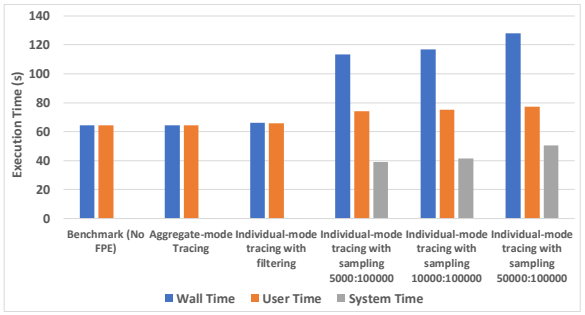

Figure 6: Overhead of FPSpy for Miniaero in various configurations.

Filtering and sampling: Capturing trace data for every floating point instruction that produces any of the events of Figure 3 is possible, but this can produce a huge data volume and considerable slowdown. In particular, the Inexact event (rounding) is common in floating point programs—rounding is normal and expected behavior in float point code and not necessarily a sign of a problem. To let the user make their own tradeoff between capture overhead and fidelity, we provide a range of sampling and filtering options.

The user can provide a subset of event types (from Figure 3) that they want to capture. By setting the floating point exception masks to correspond, FPSpy will then only incur an overhead if one of the events in the subset occurs. No overhead is incurred for other events. We use this later to capture all events except Inexact (rounding), in order to focus on lower incidence events that are much more likely to indicate a serious problem.

The user can elect to subsample the stream of events, as well as to place a limit on how many are recorded. Subsampling simply means skipping the recording of events. For example, the user can select that only every 10th event is to be recorded, and only a million events shall be recorded. Now, after 10 million faulting instructions are observed, FPSpy will disable itself and have no more overhead.

Often, however, we do want to capture all types of events, including Inexact, and we want to do so across the entire execution of the application. FPSpy includes a Poisson sampler for this purpose. The Poisson sampler repeatedly enables and disables FPSpy during the execution of the application, with the length of each on (and off) period being drawn from an exponential distribution with a user-selected mean. Time can be accounted for as virtual time (essentially instructions executed), or real time. This mechanism allows us to implement semantics like "use FPSpy 5% of the time the program is executing", with the periods of time during which FPSpy is active forming a valid random sample via the PASTA principle.

3.7 Overhead and scaling

FPSpy's overhead varies considerably depending on the mode used, which events are being tracked, and the occurrence rate of those events in the program being measured.

Regardless of the program being studied, aggregate-mode always has virtually no overhead. Here, the only interactions with FPSpy occur at program startup and shutdown, and these require milliseconds. In contrast, individual-mode operation can have tremendous overhead. In the worst case for individual-mode, if no filtering or sampling is done, every faulting instruction can result in a several thousand cycle overhead as FPSpy interacts with the kernel. The

overhead is limited to the context of a faulting instruction.

By far, the biggest practical concern in individual-mode is tracking Inexact events (rounding). If tracking rounding is not needed, the overhead is typically only slightly higher than in aggregate-mode. Recall that rounding is typically an *expected* event that is a normal element of a floating point operation and thus occurs frequently. The other events are, to varying degrees, more likely to be a sign of a problem, and thus are typically much more rare. If tracking rounding is needed, sampling can reduce the overhead.

Figure 6 shows an example of the overhead of operation of FPSpy. Here, Miniaero was run under FPSpy under each of the modes and settings used later in this paper, under two more intrusive settings, and with no FPSpy. The details of the hardware and this benchmark are given in Section 4. For each setting, Miniaero's wall-clock time, as well as system and user time are shown. As can be seen, aggregate-mode, and individual-mode without capturing Inexact (rounding), have virtually no overhead. This is true across our benchmarks and applications.

The last three configurations shown in Figure 6 are for individual-mode, with Inexact (rounding) captured, using Poisson sampling of 5% (as in Section 4), and higher rates of 10% and 50%. The highest overhead is 2x, with a major component being system time due to kernel interaction. Recall that in individual mode, each floating point event that occurs involves two faults into the kernel (the floating point exception and the single step trap) as well as matching signal deliveries from the kernel to the user space FPSpy tool. This expands the cost of a floating point instruction that causes an event from a small number of cycles to thousands of cycles.

It is important to understand that the actual overhead in individual mode is *highly* dependent on what events are being captured and how frequently they occur. In particular, if Inexact (rounding) events are being captured and the program being tested produces many of them, the overhead can be much higher than 2x seen in Miniaero, with the ~1000x instruction-handling overhead described above being the limit. LAMMPS (described later), for example, had a 127x slowdown. If the user wants to capture Inexact events, sampling is often critical to keeping the overhead under control.

Each thread in the application is monitored independently, with its trace data also being written to an independent log file (or other independent target). FPSpy is "embarrassingly parallel" internally and thus will scale along with the hardware—there is a fixed overhead per thread. Consequently, scalability is limited by the file system or logging infrastructure to which we write trace data. As we scale the hardware, we have more logs to write and the overall volume of events grows, all else being the same. The only I/O operation needed is an append, and log records are designed so that ordering is not needed, even within a single thread's log.

3.8 Portability

Although we have developed FPSpy for x64, we believe it is likely to be portable to other platforms that have a compliant IEEE floating point implementation. The x64 features we use are the `%mxcsr` register and single-stepping mode. `%mxcsr` is how the x64 exposes the IEEE standard's condition codes and exception generation model. Other compliant architectures implement these aspects of the standard differently, but they do implement them. For example, the ARM equivalent of `%mxcsr` is `%fpscr` and it is even nearly identical in semantics. We use x64's single-stepping mode as a convenient, but nonessential way to place a temporary breakpoint immediately after the faulting floating point instruction—it lets us avoid determining the instruction length, which is quite complex on x64. Any other breakpoint mechanism, including simply stubbing the next instruction with an invalid opcode, which is possible on all architectures, would also work. Furthermore, on a RISC architecture, this is trivial to do because of the fixed instruction length.

Name	Dependencies	Problem	Exec Time
Application/Frameworks			
Miniaero	Kokkos [2]	Example	1m 4.420s
LAMMPS	MPI	Methane Forces	76m 2.785s
LAGHOS	hypre [17], METIS [25], MFEM [1], MPI	Sedov Blast	116m 17.087s
MOOSE	PETSc [3], libmesh	Transient	54.275s
WRF	NetCDF [39], MPI	Squall2D_y	30m 25.019s
ENZO	MPI, HDF5 [47]	GalaxySimulation	26m 37.805s
GROMACS	MPI, MKL, OpenMP	1AKI in Water	221m59.184s
Benchmarks			
PARSEC 3.0	GSL [20], Intel TBB [36]	Simlarge	2m30.178s
NAS 3.0	N/A	Problem Size 1	4m50.443s

Figure 7: Applications and benchmarks in study comprise ~7.5M LOC, multiple languages, and threading, OpenMP, and MPI.

Is FPSpy portable to GPUs? This depends on compliance with the IEEE standard. For example, it is our understanding that current NVIDIA GPUs support neither the floating point condition codes nor raising an exception when a condition code is set. This was also true in the past for AMD and Intel GPUs, but AMD's Vega architecture has support for floating point condition codes and exceptions. In order to work in a noncompliant environment, an FPSpy-like tool would need to take an entirely different approach, probably a compile-time transform or binary rewriting scheme to introduce software guards. FPChecker [26] claims to be the first tool to do the former for GPUs.

4 Study design

Using FPSpy and other methods, we created a methodology for studying application. The Ubuntu-default gcc 5.4 toolchain was used. Unless otherwise noted, all testing was conducted on a Dell R815, which sports four 16 core 2.1 GHz AMD Opteron 6272 processors and 128 GB of RAM split among 8 NUMA zones. These processors support the SSE4.2 and AVX floating point instruction sets. The machine runs Ubuntu 16.04 with 4.4.0 kernel.

Applications and benchmarks: Figure 7 summarizes our target applications and benchmarks, their dependencies, the specific example problem we ran, and the execution time, unencumbered by FPSpy. A range of concurrency models were used, each being the recommended model for a single-node environment. This includes configurations from a single thread, to multiple threads, to multiple MPI processes, sometimes with multiple threads per process.

Our applications form a very large dataset, comprising millions of lines of code with complex dependencies. Five of them are in the top-20 most commonly used applications run on NSF's XSEDE resources [43]. Miniaero is a Mantevo [14] miniapp (one of several used for evaluation of supercomputing environments by Sandia National Labs) that solves the compressible Navier-Stokes equation. Miniaero is written in C++ and C and contains about 4400 lines of code. Miniaero is dependent on kokkos for OpenMP and Pthreads.

LAMMPS [37] is a molecular dynamics simulator primarily for materials modeling. It is primarily written in C++, although it contains some Tcl and Fortran. LAMMPS has about 1.3 million lines of code, and depends on an MPI library, such as OpenMPI. LAGHOS [16] is a hydrodynamics application that solves the time-dependent Euler equations of compressible gas dynamics using finite element analysis. LAGHOS is written almost exclusively in C++ and contains 25,000 lines of code. LAGHOS depends on hypre for linear solving, MFEM for meshing, and METIS for graph partitioning, as well as an MPI library. MOOSE [21] is a parallel finite element framework with the ability to solve mechanics, phase-field, Navier-Stokes, and heat conduction problems. Its codebase contains about 1.2 million lines of C++, Python, and C. MOOSE depends on PETSc for partial differential equation solving, as well as libmesh for meshing. WRF [38, 44] is a weather forecasting tool used by NOAA for hurricane prediction and storm forecasting. WRF is primarily written in Fortran and C, and contains about 1.4 million lines of code. WRF depends on NetCDF for array data structures, as well as on an MPI library. ENZO [13] is an astrophysics and hydrodynamics simulator. ENZO is written in C, Fortran, and Python and contains about 307,000 lines of code. ENZO depends on HDF5 for data storage, as well as an MPI library. GROMACS [4] is a molecular dynamics application primarily concerned with lipids, proteins and nucleic acids. GROMACS is written in C++ and C and contains about one million lines of code. GROMACS depends on an MPI library, an MKL library, like OpenBLAS or Intel MKL, and OpenMP.

We also consider two benchmark suites. PARSEC [11] is a set of benchmarks often used to test compiler optimizations and architectural concepts. PARSEC is written in C and C++ and contains 3.5 million lines of code spread over its various benchmarks. PARSEC depends on the GNU Scientific Library as well as Intel's Threading Building Blocks for Parallel Programming. NAS 3.0 [5, 6, 24] is a set of small programs developed by NASA to benchmark parallel computing. NAS 3.0 is written in Fortran and C and contains about 21,000 lines of code. NAS 3.0 has no external dependencies.

Our study consisted of the following passes run over the applications and benchmarks.

Source code analysis: Here we used grep, cscope, and similar tools to examine the source code for invocations of functions FPSpy needs to intercept. These include `fork()`, `clone()`, `pthread_create()`, `pthread_exit()`, `signal()`, `sigaction()`, `uc_mcontext` structures (used for manipulating register state during signals), `SIG*` macros for referring to signals, and the `fe*` unctions and `FE_*` macros for accessing and manipulating the FPU control state and information.

It is important to understand the limitations of this analysis. First, it does not delve beyond the immediate source code of the application. That is, if a shared library or wrapper uses these functions, we do not see it. Finally, our analysis does not consider inline or other assembly code within the source code that directly manipulates the floating point control/status state, for example by explicitly using the `ldmxcsr`/`stmxcsr`, `fxsave`/`fxrstor`, etc, instructions.

Aggregate-mode tracing: Here we run the application under FPSpy, with FPSpy configured to operate in aggregate-mode. This attempts to capture, for each thread and process of the application, the set of floating point events encountered throughout its execution. We ran our tests using both aggressive and non-aggressive operation with identical results. For all benchmark using floating point, other than WRF, there is no difference.

Aggregate-mode tracing produces a low volume of data, and has virtually no overhead. Our purpose with this part of the study is to discern whether an application exhibits problematic behavior at all.

Individual-mode tracing with filtering: In this step, we run each application under FPSpy, with FPSpy configured to operate in individual-mode. The filtering and sampling mechanisms are configured to capture every instruction in every thread or process that produces a floating point event other than Inexact. In other words, we find every instruction that produced a possible problem other than rounding.

This part of the study produces moderate volumes of data, and has low overhead across the board. Recall that overhead is only incurred when an instruction faults. Our purpose here is to find the specific instructions generating the most problematic events, particularly Invalids, Overflows, and Underflows.

Individual-mode tracing with sampling: In this final step, we run each application under FPSpy, with FPSpy configured to operate in individual-mode. The filter and sampling mechanisms are set such that every event type, *including* Inexact (rounding) is captured. The Poisson sampler is enabled and configured with 5% coverage. More specifically, we configure the Poisson sampler with a 5ms mean on time, and a 100ms mean off time, using virtual time.

Even with sampling, this part of the study has considerable overhead and produces large volumes of data. About 2 TB of trace data were captured during runs that took about a week. Note that the vast majority of this data concerns Inexact (rounding) events. Our purpose is to capture the distribution of the kinds of instructions that produce rounding with an eye to designing a future system to manage rounding differently in such applications.

5 Study results

We now describe the results of applying the FPSpy-based methodology to the applications and benchmarks. It is important to point out that the occurrences of problematic floating point events are just that: problematic. Ground truth is not known to any bug-finding/hardening tool. It would constitute either having a bug-free program or foreknowledge of the results of a bug-free program. As a consequence, the existence of an event does not imply an incorrect output from the program. Nonetheless, events such as Invalid, DivideByZero, and Overflow are suggestive of problems.

Validation: To validate FPSpy before using it in our methodology, we built a range of test programs that produce all of the events FPSpy can detect, within different execution models (single process/thread, single process/multiple thread, multiple processes, multiple processes each with multiple threads, and confounding all with signals). FPSpy passed these tests, producing outputs that correspond to what was constructed.

5.1 Use of floating point control is rare

Figure 8 shows whether each application or benchmark uses mechanisms that FPSpy intercepts or otherwise manipulates. With respect to the generality of FPSpy, we are primarily concerned about uses of functions, macros, and syscalls that would require FPSpy to disable itself in order to preserve application semantics. These are the `fe*` family of floating point control functions. Essentially, anything

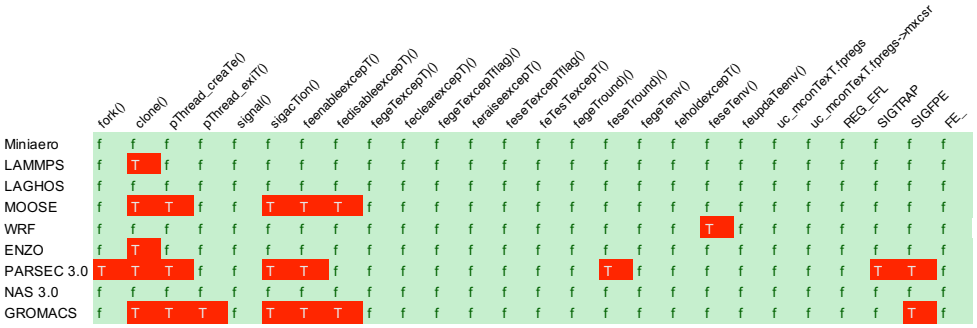

Figure 8: Source code analysis. A "T" (red square) indicates the source code uses the noted mechanism.

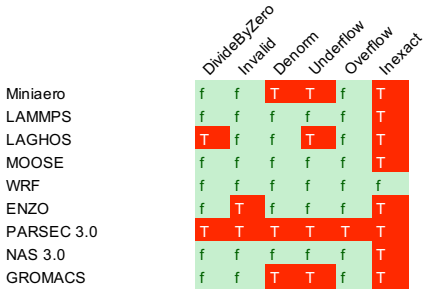

Figure 9: Analysis of aggregate-mode tracing of applications. A "T" indicates that the event occurred. Clearly, problematic events do occur during the execution of these programs.

from `feenableexcept()` rightwards in the figure would cause FPSpy to disable itself. Anything from `sigaction()` leftwards can be handled by FPSpy without issue.

It is important to understand that this figure results from a static analysis of the source code. In fact, what matters is whether the code is encountered *dynamically*. Although MOOSE, WRF, some PARSEC benchmarks, and GROMACS contain uses of floating point control mechanisms, only WRF actually executes any of them in our testing. As a consequence, the WRF results (which we do include), show no events occurring—FPSpy stepped aside in that specific case. For all other applications and benchmarks in our study, FPSpy observed the behavior from start to finish.

Beyond arguing for FPSpy's generality, the static and dynamic results also suggest something important about the applications: the use of floating point control during execution is rare. Because very few applications use any floating point control, problematic events, especially those other than rounding, may remain undetected. The results seem to support the previous study of developers [15], suggesting that many developers are not prepared to mitigate floating point error, whether in development or execution.

5.2 Problematic events occur in practice

Many of our applications and benchmarks produce potentially problematic floating point events. Figure 9 shows the results of using FPSpy in aggregate mode on our applications and benchmarks. Only LAMMPS, MOOSE, and the NAS benchmarks operate without any concerning results.[2] At the other extreme, ENZO produces

[2]Recall that Inexact (rounding) events are expected in normal operation.

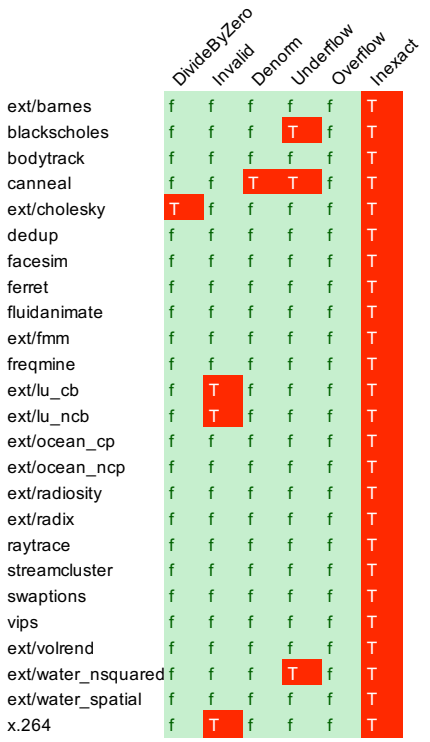

Figure 10: Analysis of aggregate-mode tracing of PARSEC benchmarks. A "T" indicates that the event occurred.

NaNs, while LAGHOS divides by zero. Underflow events (Miniaero, LAGHOS, PARSEC, GROMACS) are relatively common, but not necessarily indicative of a problem. Denorm events (Miniaero, PARSEC, GROMACS) are similar.

5.3 Benchmarks may be unrepresentative

The PARSEC benchmarks produce every single event (on a different problem size, it did not produce an Overflow). Figure 10 breaks down the PARSEC results by individual benchmark. PARSEC contains benchmarks inspired from a range of domains, not just scientific computing. It is nonetheless disconcerting to see the most severe event, Invalid, occur in LU decompositions, and the less severe DivideByZero occur in a Cholesky decomposition. These are a part of the "external" benchmarks, however, derived from SPLASH.

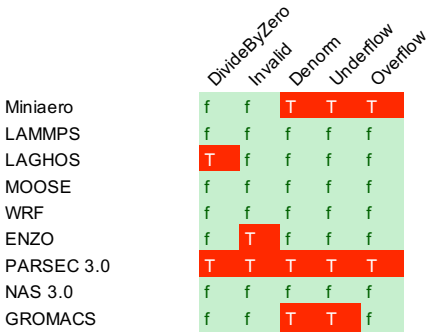

Figure 11: Analysis of individual-mode tracing with filtering of applications and benchmarks. Full instruction coverage (every faulting instruction captured) but Inexact event not tracked. A "T" indicates that at least one instruction encountered the event.

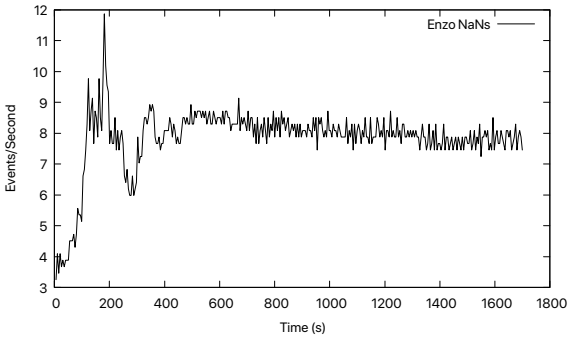

Figure 12: Rate of Invalid events over time in ENZO, captured using individual-mode tracing with filtering.

In contrast, all of the NAS benchmarks behave well. One could argue that the PARSEC benchmarks, with their various problematic events, are perhaps more representative of real world applications.

5.4 Instruction-level detail

Figures 11 through 14 follow up with individual-mode tracing. Recall that individual-mode tracing captures potentially every faulting instruction and thus is much slower than aggregate-mode tracing. In Figure 11 we use filtering to limit this cost. We capture all faulting instructions, except those that fault due to rounding. The raw data in the trace, which includes the address of the instruction, allows a developer to trace faults back to specific source lines of code.

Individual-mode tracing also allows us to see the temporal behavior of problematic events. For example, Figure 12 shows the rate of Invalid Events in ENZO over time—NaNs occur throughout most of execution. Figure 13 zooms in to a 3 second interval during which bursts of DivideByZero events occur in Laghos.

In Figure 14 we use sampling to limit the cost, but we include Inexact events. Note that WRF displays rounding behavior in Figure 14, but not when run with aggregate-mode, in Figure 9. Because aggregate-mode relies on the sticky nature of floating point exceptions in the `%mxcsr` register, aggregate-mode is unable to detect the rounding events, as WRF clears the register during its own floating point control. However, individual-mode sampling is able to capture the events, as it captures them as they arise. However,

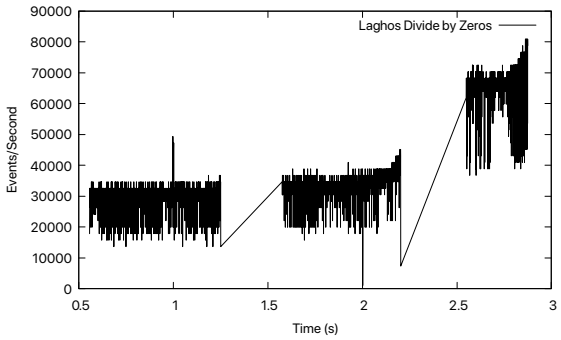

Figure 13: Bursts of DivideByZero events in LAGHOS, captured using individual-mode tracing with filtering.

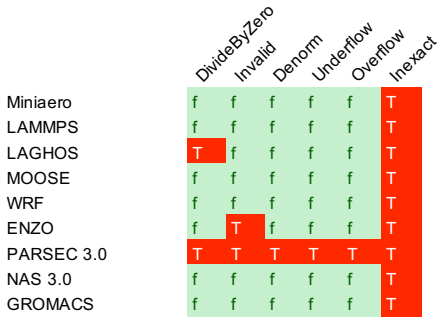

Figure 14: Analysis of individual-mode tracing with sampling of applications and benchmarks. 5% sampling, including Inexact. Poisson sampling with 5000 us mean on time and 100000 us mean off time using virtual timer. A "T" indicates that at least one instruction encountered the event.

sampling has caused us to miss the Underflow and Overflow events captured with the previous method.

5.5 Caveats

It is important to understand that we have tested benchmarks and *examples* from large application frameworks on a single compiler and a single machine. There are natural questions to ask: How input-dependent are our results? How dependent are the results for the application frameworks on their use cases? How dependent are the results on the compiler and/or architecture? These are not questions that we can answer with this study. We also do not know ground truth here, so we cannot make claims about whether our codes are "correct" or not. The study is telling us two things: (1) FPSpy seems to work and fulfill the requirements laid out in Section 2. (2) *Potentially* problematic behavior occurs in *some* of the codes.

6 Evaluating prospects for rounding mitigation

We now turn from using FPSpy's traces to find problematic floating point behavior (and thus potential bugs that could perturb results), and instead use them as enabler of system design.

Inexact (rounding) events are a normal part of floating point arithmetic. Nonetheless, they reflect a loss of precision in the computation that the developer does need to reason about to assure reasonable results. In effect, losses of precision introduce errors into the computation. When modeling, for example, a system that

Name	Total Inexact Events	Inexact Events/sec
Miniaero	6,287,103	1,108,944
LAMMPS	394,272,578	67,851
LAGHOS	359,200,445	650,000
MOOSE	119,050,994	1,445,125
WRF	6,299,892	65,543
ENZO	402,709,981	222,227
GROMACS	16,444,453	26,224

Figure 15: Inexact event count and rate for each application.

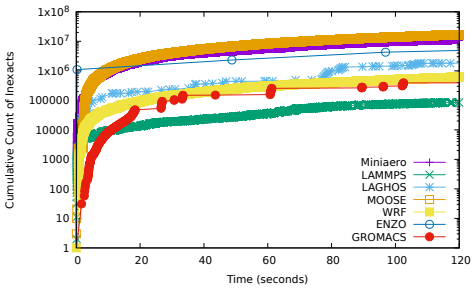

Figure 16: Cumulative Inexact (rounding) events for the first two minutes of execution of each application.

involves chaotic dynamics, such errors, even if they are tiny, can result in diverging or incorrect solutions.

In ongoing work, we are exploring how to control or mitigate such losses of precision using techniques implemented at the runtime, compiler, kernel, and hardware levels. Understanding the workload—the properties of instructions that round in real applications—is important to such work. FPSpy's individual-mode tracing, specifically with Poisson sampling, allows us to capture workloads and study them. The traces summarized in Figure 14 provide about 2 TB of such workload. Figure 15 shows the number of Inexacts captured via this methodology, and their overall rates, while Figure 16 shows how Inexacts accumulate over time.

One potential approach to a rounding mitigation system is the use of trap-and-emulate processing (as in FPSpy or a virtual machine monitor) or dynamic binary patching [29] to bridge between floating point instructions that command the x64 hardware floating point unit, and calls into an arbitrary precision software floating point unit such as MPFR [19]. This would allow existing, unmodified application binaries to seamlessly execute with higher precision as necessary, resulting in less or even no rounding.

A first-line question for the prospects of such a system is the nature of locality for the instructions that round. Without locality, such a system cannot work because the overheads involved with instruction decoding or binary patching cannot be amortized. We can answer the question using FPSpy's traces. Characterizing the traces via rank-popularity statistics is particularly useful and enlightening.

Figure 17 shows rank-popularity distributions with respect to the faulting instruction's form (e.g., add, multiply, divide, etc) for each trace. We only differentiate the applications (bold lines) from the PARSEC and NAS benchmarks, and our primary concern is with the applications. The graph shows that even for the most extreme application, fewer than 45 instruction forms are used. 20 or fewer instruction forms are used by each of the remainder of the

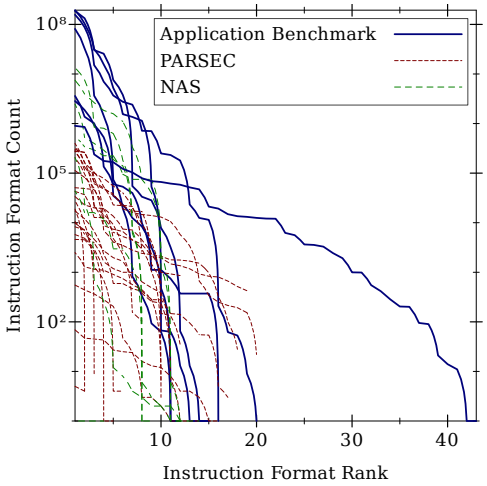

Figure 17: Rank-popularity of rounding instruction form. A small number of instruction forms account for all rounding.

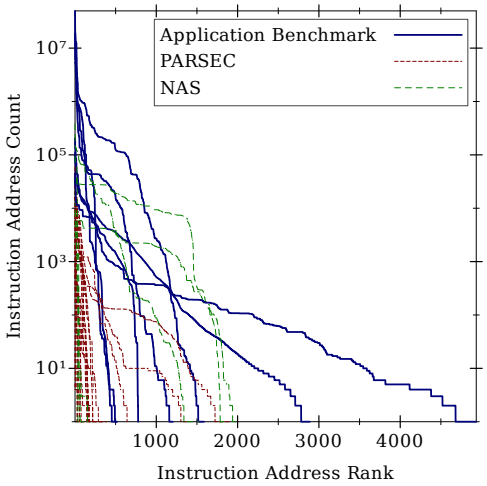

Figure 19: Rank-popularity of rounding instruction address. A small number of addresses account for all rounding.

applications (as well as the NAS and PARSEC benchmarks). There is also clearly a heavy skew in the rank-popularity distribution (note that the vertical axis is on a log scale). For the most part, for each code, fewer than 5 instruction forms cover >99% of the instructions that encounter rounding.

Figure 18 digs deeper. 39 instruction forms constitute the total coverage of observed instructions that round across every sampled code other than GROMACS. The figure is a histogram showing the number of applications or benchmarks that use each of these forms. GROMACS uses 25 forms not used by any other code, in addition to 16 forms used by other codes.

Figure 19 shows rank-popularity distributions with respect to the rounding instruction's address. In the most extreme case, <5000 instructions in the code account for all the rounding that was encountered. More commonly, <2000 instructions are sufficient. As with the instruction form analysis, the rank-popularity distributions are also heavily skewed. For the most part, <100 instructions account for >99% of the rounding events.

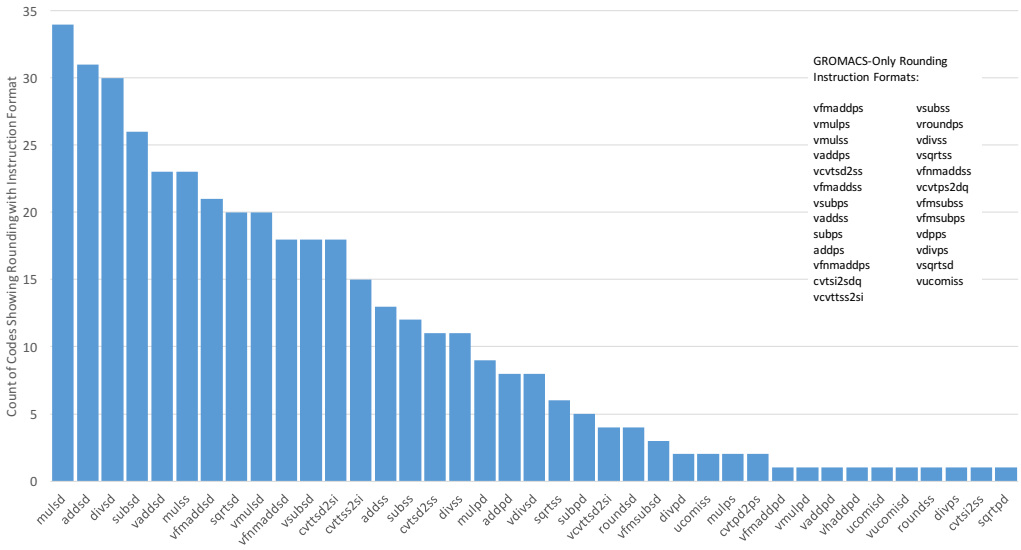

Figure 18: Rank-popularity of specific instruction forms among the codes used in the study. 39 instruction forms completely cover all the codes other than GROMACS, which uses 25 forms not seen elsewhere.

A rounding mitigation system could leverage these low limits and skewed distributions. By focusing on <5000 instruction sites and handling <45 instruction forms at those sites, such a system could radically change the effects of rounding on an application.

7 Related work

Scientific openness and reproducibility first became a hot topic within the social sciences [35], while the reproducibility of *computational* research has recently become a major concern within the *physical sciences* via the work of Stodden and others [45, 46]. Computational research has a complete reliance on floating point arithmetic, as defined in the IEEE 754 [22], and 754-2008 [23] standards. Many physical systems and their models exhibit chaotic dynamic behavior [32] and thus are particularly sensitive to perturbation by floating point issues. A study of developer understanding of floating point [15] suggests that such issues might be more common than expected. A floating point issue has already proven to explain divergent results for a port of NCAR's CESM climate model [31].

Program analysis tools have a long history. FPSpy is intended to have the same ease of use as the commonly employed strace and ltrace [12] tools. Coverity [10] and Fortify are perhaps the most widely used static analysis tools, but do not focus on floating point and require recompilation of the target application. Valgrind [33] is a widely used dynamic analysis tool framework, which involves binary rewriting. Within that framework, the Verrou [18] tool is closest to FPSpy in that it hunts for floating point rounding issues.

A range of tools have been developed to improve floating point code within applications. For example, FpDebug [9] searches for situations in which rounding errors produce divergent results. Other tools find situations where cancellations cause bogus outputs [27], detect numeric instability at run-time [7, 28], or find the lowest precision that does not produce apparent divergent outputs [40].

FPSpy fits into the thread of work that tries to find floating point issues, and then identify their root cause in the software, or ameliorate them more directly. The closest work is Milroy et al [30], Herbie [34], Herbgrind [41], and Flit [8, 42]. Milroy et al and Flit use variant compilation to induce output variation, and then localize the source of this variation to a root cause in the source. Herbgrind also attempts to find the root cause, although here the starting point can be instructions that have high rounding error. Herbie optimizes floating point source code expressions with the goal of increasing accuracy across all inputs. In contrast to these systems, the goal of FPSpy is to track hardware-visible floating point events in existing, unmodified binaries without programmer or user cooperation, possibly even in a deployment.

8 Conclusions

We described FPSpy, a tool for tracking potentially problematic floating point events occurring during the normal execution of an existing, unmodified x64 application binary on Linux. FPSpy's overhead can be throttled, allowing a tradeoff between which and how many unusual events are to be captured, and the slowdown incurred by the application. FPSpy can produce traces of problematic events at the instruction granularity. Applying an FPSpy-based methodology to a range of applications and benchmarks comprising millions of source lines of code caught a collection of issues. We also used FPSpy to characterize the instructions that round within our test suite. This characterization suggests that a trap-and-emulate approach to integrating higher precision is feasible. We are currently implementing various approaches, including trap-and-emulate.

References

[1] [n.d.]. MFEM: Modular Finite Element Methods Library. mfem.org.

[2] 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[3] Shrirang Abhyankar, Jed Brown, Emil M Constantinescu, Debojyoti Ghosh, Barry F Smith, and Hong Zhang. 2018. PETSc/TS: A Modern Scalable ODE/DAE Solver Library. *arXiv preprint arXiv:1806.01437* (2018).

[4] Mark Abraham, Teemu Murtola, Roland Schulz, Szilard Pall, Jeremy Smith, Berk Hess, and Erik Lindahl. 2015. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*

1 (07 2015).
[5] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinksi, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. 1994. *The NAS Parallel Benchmarks (NAS 1).* Technical Report RNR-94-007. NASA.
[6] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. 1991. The Nas Parallel Benchmarks. *International Journal of High Performance Computing Applications* 5, 3 (Sept. 1991), 63–73.
[7] Tao Bao and Xiangyu Zhang. 2013. On-the-fly Detection of Instability Problems in Floating-point Program Execution. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA).*
[8] Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, Dong H. Ahn, Ignacio Laguna, Gregory L. Lee, and Holger E. Jones. 2019. Multi-level Analysis of Compiler-Induced Variability and Performance Tradeoffs. In *Proceedings of the 28th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2019).*
[9] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A Dynamic Program Analysis to Find Floating-point Accuracy Problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*
[10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commuications of the ACM* 53, 2 (February 2010), 66–75.
[11] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.
[12] Rodrigo Rubira Branco. 2007. Ltrace Internals. In *Proceedings of the Ottawa Linux Symposium.*
[13] G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman, B. Smith, R. P. Harkness, J. Bordner, J.-h. Kim, M. Kuhlen, H. Xu, N. Goldbaum, C. Hummels, A. G. Kritsuk, E. Tasker, S. Skory, C. M. Simpson, O. Hahn, J. S. Oishi, G. C. So, F. Zhao, R. Cen, Y. Li, and The Enzo Collaboration. 2014. ENZO: An Adaptive Mesh Refinement Code for Astrophysics. *The Astrophysical Journal* 211, 2, Article 19 (March 2014), 19 pages. arXiv:astro-ph.IM/1307.2265
[14] Paul Crozier, Heidi Thornquist, Robert Numrich, Alan Williams, H. Edwards, Eric Keiter, Mahesh Rajan, James Willenbring, Douglas Doerfler, and Michael Heroux. 2009. *Improving performance via mini-applications.* Technical Report SAND2009-5574. Sandia National Laboratories. https://doi.org/10.2172/993908
[15] Peter Dinda and Conor Hetland. 2018. Do Developers Understand IEEE Floating Point?. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018).*
[16] V. Dobrev, T. Kolev, and R. Rieben. 2012. High-Order Curvilinear Finite Element Methods for Lagrangian Hydrodynamics. *SIAM Journal on Scientific Computing* 34, 5 (2012), B606–B641.
[17] Robert D. Falgout and Ulrike Meier Yang. 2002. Hypre: A Library of High Performance Preconditioners. In *Proceedings of the International Conference on Computational Science-Part III (ICCS '02).* Springer-Verlag, Berlin, Heidelberg, 632âĂŞ641.
[18] François Févotte and Bruno Lathuilière. 2016. VERROU: Assessing Floating Point Accuracy Without Recompiling. working paper or preprint.
[19] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Transactions on Mathematical Software (TOMS)* 33, 2 (June 2007).
[20] Mark Galassi, Jim Davies, James Theiler, Brian Gough, and Gerard Jungman. 2009. *GNU Scientific Library - Reference Manual, Third Edition, for GSL Version 1.12 (3. ed.).*
[21] Derek R. Gaston, Cody J. Permann, John W. Peterson, Andrew E. Slaughter, David Andrš, Yaqi Wang, Michael P. Short, Danielle M. Perez, Michael R. Tonks, Javier Ortensi, Ling Zou, and Richard C. Martineau. 2015. Physics-based multiscale coupling for full core nuclear reactor simulation. *Annals of Nuclear Energy* 84 (2015), 45–54.
[22] IEEE Floating Point Working Group. 1985. IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985* (1985).
[23] IEEE Floating Point Working Group. 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70.
[24] H. Jin, M. Frumkin, and J. Yan. 1999. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance (NAS 3).* Technical Report NAS-99-011. NASA.
[25] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (Dec. 1998), 359âĂŞ392.
[26] I. Laguna. 2019. FPChecker: Detecting Floating-Point Exceptions in GPU Applications. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 1126–1129.
[27] Michael O. Lam, Jeffrey K. Hollingsworth, and G.W. Stewart. 2013. Dynamic floating-point cancellation detection. *Parallel Comput.* 39, 3 (2013), 146–155.
[28] Wen-Chuan Lee, Tao Bao, Yunhui Zheng, Xiangyu Zhang, Keval Vora, and Rajiv Gupta. 2015. RAIVE: Runtime Assessment of Floating-point Instability by Vectorization. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*
[29] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools With Dynamic Instrumentation. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementatin (PLDI).*
[30] Daniel Milroy, Allison Baker, Dorit Hammerling, and Youngsung Kim. 2019. Making Root Cause Analysis Feasible for Large Code Bases: A Solution Approach for a Climate Model. In *Proceedings of the 28th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2019).*
[31] Daniel J. Milroy, Allison H. Baker, Dorit M. Hammerling, John M. Dennis, Sheri A. Mickelson, and Elizabeth R. Jessup. 2016. Towards Characterizing the Variability of Statistically Consistent Community Earth System Model Simulations. *Procedia Computer Science* 80, C (June 2016), 1589–1600.
[32] Francis C. Moon. 1992. *Chaotic and Fractal Dynamics: An Introduction for Applied Scientists and Engineers.* John Wiley and Sons, Inc.
[33] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 89–100.
[34] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*
[35] Harold Pashler and Eric-Jan Wagenmakers. 2012. Editors' Introduction to the Special Section on Replicability in Psychological Science: A Crisis of Confidence? *Perspectives on Psychological Science* 7, 6 (2012), 528–530.
[36] Chuck Pheatt. 2008. Intel Threading Building Blocks. *J. Comput. Sci. Coll.* 23, 4 (April 2008), 298.
[37] Steve Plimpton. 1995. Fast Parallel Algorithms for Short-range Molecular Dynamics. *J. Comput. Phys.* 117, 1 (March 1995), 1–19.
[38] Jordan G. Powers, Joseph B. Klemp, William C. Skamarock, Christopher A. Davis, Jimy Dudhia, David O. Gill, Janice L. Coen, David J. Gochis, Ravan Ahmadov, Steven E. Peckham, Georg A. Grell, John Michalakes, Samuel Trahan, Stanley G. Benjamin, Curtis R. Alexander, Geoffrey J. Dimego, Wei Wang, Craig S. Schwartz, Glen S. Romine, Zhiquan Liu, Chris Snyder, Fei Chen, Michael J. Barlage, Wei Yu, and Michael G. Duda. 2017. The Weather Research and Forecasting Model: Overview, System Efforts, and Future Directions. *Bulletin of the American Meteorological Society* 98, 8 (2017), 1717–1737.
[39] R. Rew and G. Davis. 1990. NetCDF: an interface for scientific data access. *IEEE Computer Graphics and Applications* 10, 4 (1990), 76–82.
[40] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing).*
[41] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding Root Causes of Floating Point Error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*
[42] G. Sawaya, M. Bentley, I. Briggs, G. Gopalakrishnan, and D. H. Ahn. 2017. FLiT: Cross-platform floating-point result-consistency tester and workload. In *Proceedings of the 2017 IEEE International Symposium on Workload Characterization (IISWC).* 229–238.
[43] Nikolay A. Simakov, Joseph P. White, Robert L. DeLeon, Steven M. Gallo, Matthew D. Jones, Jeffrey T. Palmer, Benjamin D. Plessinger, and Thomas R. Furlani. [n.d.]. A Workload Analysis of NSF's Innovative HPC Resources Using XDMoD. *CoRR* abs/1801.04306 ([n. d.]).
[44] C. Skamarock, Bogumila Klemp, Jimy Dudhia, Olivia Gill, Dale Edwin Barker, Gintaras Kazimieras Duda, Xiang-Yu Huang, Wei Wang, and Gregory N. Powers. 2008. *A Description of the Advanced Research WRF Version 3.* Technical Report SAND2009-5574. National Center For Atmospheric Research.
[45] Victoria Stodden, Peixuan Guo, and Zhaokun Ma. 2013. Toward Reproducible Computational Research: An Empirical Analysis of Data and Code Policy Adoption by Journals. *PloS One* 8, 6 (2013).
[46] Victoria Stodden, Marcia McNutt, David H. Bailey, Ewa Deelman, Yolanda Gil, Brooks Hanson, Michael A. Heroux, John P.A. Ioannidis, and Michela Taufer. 2016. Enhancing reproducibility for computational methods. *Science* 354, 6317 (December 2016).
[47] The HDF Group. 1997-NNNN. *Hierarchical Data Format, version 5.* http://www.hdfgroup.org/HDF5/.