NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2025-27

July, 2025

Architecture-independent Floating Point Spying and an Architecture for Floating Point Spying

Karl Hallsby, Liam Strand, Nick Wanninger,
Nadharm Dhiantravan, Peter Dinda

Abstract

Floating point instructions in existing, unmodified application binaries can be monitored, at the hardware level, for signs of trouble. Events such as NaNs, overflows, underflows, denorms and unexpected rounding can be readily detected, and when they do not occur, there is no overhead. We describe how we have extended the open source FPSpy tool, which provides such monitoring on x64 hardware, to be architecture-independent, with support extended to the ARM and RISC-V architectures. We also extend the open RISC-V architecture and its SonicBOOM implementation to be particularly conducive to monitoring. First, we introduce floating point traps as an architectural extension, making it possible to do monitoring in the first place. By default, this delivers signals via the kernel as is common on other architectures. Second, we introduce precise pipelined exceptions (PPEs), which allow the RISC-V architecture to deliver instruction exceptions, such as floating point traps, directly to user-space handlers, greatly reducing their overhead (by a factor of 30) by bypassing the kernel. This reduces the cost of detecting and logging an event-causing instruction in FPSpy by a factor of 3. We present detailed performance results of the architecture-independent FPSpy on our augmented RISC-V platform.

This effort was partially supported by the United States National Science Foundation (NSF) under awards CNS-2211315, CCF-2119069, and CNS-2211508.

Keywords

IEEE 754, RISC-V, floating point arithmetic, floating point hardware, computer architecture, correctness, software development

Architecture-independent Floating Point Spying and an Architecture for Floating Point Spying

Karl Hallsby
Northwestern University
Evanston, IL, USA
kgh@u.northwestern.edu

Liam Strand
Northwestern University
Evanston, IL, USA
ltrs@u.northwestern.edu

Nick Wanninger
Northwestern University
Evanston, IL, USA
ncw@u.northwestern.edu

Nadharm Dhiantravan
Northwestern University
Evanston, IL, USA

Peter Dinda
Northwestern University
Evanston, IL, USA
pdinda@northwestern.edu

Abstract

Floating point instructions in existing, unmodified application binaries can be monitored at the hardware level for signs of trouble. Events such as NaNs, overflows, underflows, denorms and unexpected rounding can be readily detected, and when they do not occur, there is no overhead. We describe how we have extended the open source FPSpy tool, which provides such monitoring on x64 hardware, to be architecture-independent, with support extended to the ARM and RISC-V architectures. We also extend the open RISC-V architecture and its SonicBOOM implementation to be particularly conducive to monitoring. First, we introduce floating point traps as an architectural extension, making it possible to do monitoring in the first place. By default, this delivers signals via the kernel as is common on other architectures. Second, we introduce precise pipelined exceptions (PPEs), which allow the RISC-V architecture to deliver instruction exceptions, such as floating point traps, directly to user-space handlers, greatly reducing their overhead (by a factor of 30) by bypassing the kernel. This reduces the cost of detecting and logging an event-causing instruction in FPSpy by a factor of 3. We present detailed performance results of the architecture-independent FPSpy on our augmented RISC-V platform.

Keywords

IEEE 754, RISC-V, floating point arithmetic, floating point hardware, computer architecture, correctness, software development

1 Introduction

There is an exploding interest in correctness in scientific applications, exemplified by the NSF and DOE's cross-cutting CS2 funding program. The correctness of floating point arithmetic is an important part of this interest, at levels ranging from numeric methods and algorithms, through languages, libraries and compilation, all the way down to hardware. In this work, we are focused on the architectural instructions executed by any existing, unmodified scientific application binary that implements the IEEE 754 standard [15, 16], which is is almost always what is meant when we use the word "floating point arithmetic" in scientific computing.

This effort was partially supported by the United States National Science Foundation (NSF) under awards CNS-2211315, CCF-2119069, and CNS-2211508.

Due to a required feature of the IEEE 754 standard, *floating point exceptions*,[1] it is possible to build tools that use the hardware to detect when a window of instructions[2] has resulted in a small set of unusual events occurring, with virtually zero overhead. These include overflow (Infinity production), underflow (zero and/or subnormal production), NaN (non-number production or consumption), divide-by-zero, and an imprecise result (due to rounding). Some hardware adds additional events, for example to make it possible to differentiate subnormals and zeros. Such events can correlate with correctness issues, some more than others. An additional optional feature, *floating point traps*,[3] allows us to *trace* individual instructions, which can be very useful when deciphering the origin of the event. Floating point traps also enable other tools that *modify* the floating point behavior of a program, for example to execute instructions at a different precision. Tools such as these are very closely coupled with hardware and the kernel, particularly if performant results are required. This makes portability a challenge.

There are two thrusts to the work described in this paper. In the first thrust, we describe our experience with extending one such tool, the open source FPSpy system, from being x64-only, to being architecture-independent, with current support on x64, ARM, and RISC-V. As part of that effort, we also describe kernel module support that reduces the overhead of using floating point traps.

Some architectures do not support floating point traps, which renders tools of interest unable to detect events at instruction granularity unless we are willing to modify the scientific application binary, either by recompilation, or static or dynamic rewriting. While floating point traps are a mandatory part of the x64 floating point specification, they are optional on ARM,[4] and *deliberately not included* in the various RISC-V specifications (i.e., F, D, Q, ...) for floating point instructions. Beyond simply having floating point

[1] Confusingly, "exception" in this standard simply means that the event is recorded, not that exceptional control flow occurs. Floating point exceptions are implemented in most hardware, including the architectures described here, as sticky condition codes in a user-visible control/status register.
[2] For example, those executing over an interval of time as marked out by timer events.
[3] "Trap" here means that exceptional control flow occurs. A typical implementation will cause the instruction to fault into the kernel, similar to an instruction causing a page fault.
[4] It may be surprising to learn that various implementations, including server-level chips like ThunderX2 and Neoverse-N1, do not support floating point traps, while even the lowest-end Apple M1 does.

traps at all, what architectural features would be desirable to make our tools of interest more useful and performant?

In the second thrust of work described in this paper, we explore how to make an architecture more amenable to tools such as FPSpy by extending the open RISC-V architecture and its SonicBOOM implementation. We introduce floating point traps into the RISC-V architecture, then explore how to deliver them as precise instruction exceptions directly to user-level code, bypassing the kernel altogether. This lowers the cost of trap delivery to the tool to be on par with that of a function call.

Our contributions are as follows:

- We describe the design and implementation of architectural-independence in the open source FPSpy system, including support for the x64, ARM, and RISC-V architectures. We believe our experiences can generalize to other such tools.
- We describe a Linux kernel module for x64 (and, in principle, other architectures) that allows much faster and lower overhead dispatch of floating point traps to user-level tools. The module is not FPSpy-specific.
- We describe how we extended the RISC-V architecture and SonicBOOM's [2, 36] implementation of the floating point F (Float), D (Double-Precision Float), and Q (Quad-Precision Float) extensions on RISC-V to support floating point traps, including controlling them as needed for FPSpy-like tools.
- We describe how we extended RISC-V firmware and Linux to support our new floating point traps, including delivery of them as `SIGFPE`s, similar to the x64 and ARM architectures.
- We describe precise pipelined exceptions (PPEs), an approach for fast dispatch of precise instruction exceptions to user-level handlers, bypassing the kernel. We implement PPEs in SonicBOOM and use them as an alternative delivery mechanism for floating point traps and breakpoint traps. Firmware and Linux support allows a user-level process to optionally request delivery in this manner.
- We integrate floating point trap and PPE support into FPSpy and describe the challenges of this integration.
- We evaluate the performance PPEs and their use in FPSpy on our augmented RISC-V platform running on FPGA-accelerated Firesim. We find that PPEs reduce event overheads by a factor of 30 and allow FPSpy to handle each event-generating floating point instruction about 3 times faster than using conventional kernel-level traps and signals.

The architecture-independent FPSpy and our RISC-V extensions for floating point traps and PPEs will be made available on publication.

2 FPSpy

FPSpy [11] is an example of the kind of tool that relies heavily on the hardware floating point implementation in order to achieve its ends with minimal overhead. In its current state, it is highly-specialized to the Intel/AMD x64 architecture, the Linux process environment, and the ELF dynamic linking model.

2.1 x64 hardware support

FPSpy and tools like it build on two specific forms of hardware support, floating point condition codes ("exceptions" in IEEE 754 terminology), and floating point traps, which both exist on x64.

Floating point exceptions: The IEEE 754 standard requires that the hardware track the following exceptions for each floating point operation:

- Inexact. The result could not be exactly represented and thus was *rounded*.
- Underflow. The result was "too small" and thus became a denormalized number or a zero.
- Overflow. The result was "too large" and thus became an infinity.
- DivideByZero. The operation attempted division by zero.
- Invalid. An input or the result was not a number (NaN).

An implementation can provide additional exceptions. x64 also detects Denorm, which can be used to differentiate between an underflow that resulted in a denormalized number from one that resulted in a zero. The presence of exceptions such as Overflow, DivideByZero, or Invalid are highly problematic in a computation. Underflow or Denorm may be a sign of a problem. Because rounding is necessarily commonplace in floating point arithmetic, Inexact is usually not a sign of a problem, although the user may want to observe these events nonetheless, or to modify instruction behavior.

According to the standard, floating point exceptions must be tracked in a "sticky" manner. On x64, these events map to condition code bits on the `%mxcsr` register. User-level code can both reset and read them. When an exception occurs, the corresponding bit is set. It then *remains set* (it is "sticky") until the user resets it.

Floating point traps: The standard also optionally provides for creating a CPU-level trap when one of these sticky condition code bits transitions from a `0` to a `1`. x64 implements this through a set of user-configurable bits on `%mxcsr` that allow the user to mask such traps. If the user unmasks the Invalid trap, for example, and zeros the Invalid condition code, then the next instruction that produces or consumes a NaN will cause a CPU-level exception (`#XF`) which is delivered to the kernel much like any other exception. The kernel then translates this into a `SIGFPE` signal that it delivers to the thread that was executing the problematic instruction.

FPSpy avoids decoding or emulating any instruction by using x64's single-step operation, called "trap mode", which is available to user-level code. When the CPU is in trap mode, each instruction generates a hardware trap (just as a breakpoint would) once it has finished executing, which the kernel translates into a `SIGTRAP` signal delivered to the thread executing the instruction.

2.2 General description

The goal of FPSpy is to "slide under" an existing, unmodified application binary of any kind and collect trace information without interfering with the binary. If the running application binary itself uses some hardware or kernel feature that FPSpy needs, FPSpy gracefully stops collecting trace information, disables itself, and "steps aside" to allow the application binary to continue. The intent behind such a graceful shutdown is to allow FPSpy to potentially be used in production environments to monitor applications as they do real work, not just during development. Many of the design choices of FPSpy are made to be able to "step aside".

Implementation: FPSpy is implemented as an `LD_PRELOAD` shared library with link-time functions that are called using dynamic linker constructor/destructor support. Constructors are used on every process creation. The constructor code overrides not only

process and thread creation functionality (i.e., FPSpy traces through / "follows" `fork()`, `exec()`, `pthread_create()`, `clone()`, etc),. but also inserts itself into a wide variety of other library functions that manipulate floating point state, signal handler introduction, etc. It does the latter to catch cases for which it should "step aside". It also installs signal handlers for a range of signals, including `SIGFPE` and `SIGTRAP`, which drive execution as described below.

Overhead/detail tradeoff: An important aspect of the design is to allow the deployer to decide on the tradeoff between overhead and trace detail. In effect, the x64 hardware support described above is always detecting the events FPSpy consumes. In trading off between overhead and detail, we are simply deciding which events the hardware will forward to FPSpy via the kernel.

At one extreme, the *aggregate mode* of operation, the overhead is effectively zero. In aggregate mode, the system simply manages the floating point condition codes in such a way that it can determine which conditions (floating point exceptions) occurred during the entire lifetime of a thread. Regardless of which exceptions are tracked and how many times they occurred, this is a constant cost of ~1 ms.

The *individual mode* of operation is where most of the tradeoff space is. Here, the system detects *individual instruction executions* that cause exceptions of interest. The user specifies which exceptions to track. FPSpy unmasks the traps for these in every thread's `%mxcsr`. When an instruction executing in a thread traps, FPSpy records a timestamp, the conditions that lead to the trap, address of the trapping instruction, the stack pointer (optionally providing information to backtrace after the fact), and an optional snapshot of the (undecoded) instruction, all in a per-thread trace.

The overhead of FPSpy when running in individual mode depends on which exceptions are being tracked and how many of those exceptions actually occur during execution. If the deployer selects only a small set of exceptions that indicate severe problems (e.g., setting just Overflow, DivideByZero, and NaN), then we would expect virtually no overhead when running an unproblematic application binary. If those severe problems are commonplace, the overhead is much higher, but perhaps understandably so. If, for example, the binary "generates a NaN" and that NaN propagates, many Invalid exceptions will occur, and FPSpy will record them all.

Generally, as the deployer chooses to track more exceptions, the overhead gets higher. At the other extreme, if the deployer chooses to track Inexact, the volume of traps and thus the overhead will be maximized since rounding is common in floating point code even when it is working correctly. To better support use cases that observe rounding and similar high volume exceptions, FPSpy includes numerous sampling options which allow a further tradeoff to be made between the overhead and the probability of detecting an exception. All of these are already architecturally independent.

Manipulation: FPSpy has limited support for manipulating the floating point state to try to uncover problematic application behavior. These include the ability to force the hardware to use specific rounding modes, and to manipulate the denorm behavior. Rounding modes and the ability to manipulate them are specified in the standard. FPSpy implements this support by forcing `%mxcsr` manipulations. Non-standard, but increasingly common is the ability to force the hardware to increase performance by converting denormalized results to zero (FlushToZero (FTZ)) and/or treating denormalized inputs as zero (DenormsAreZero (DAZ)). FPSpy can retroactively change this behavior, simulating how well the code might work on an environment in which denorms and thus "gradual underflow" are not implemented (e.g., GPUs).

2.3 Individual mode engine

Most of the challenges involved in making FPSpy architecture-independent (and in providing necessary architectural support for FPSpy) lie in the requirements of the individual mode of operation. Making aggregate mode architecture-independent is essentially just an afterthought. Hence, it is worth understanding how individual mode works in more detail.

For each thread, FPSpy implements a four-state state machine. Two of these states, INIT and ABORT, exist only to simply startup and graceful "stepping aside". For the most part, we will alternate between the AWAIT_FPE and AWAIT_TRAP states, and FPSpy will be triggered by the `SIGFPE` and `SIGTRAP` signals.

When AWAIT_FPE is entered, the tracked floating point exceptions have been zeroed, and the tracked floating point traps will have been unmasked. The application thread is now executing as normal. Eventually, it will hit a floating point instruction that raises a tracked exception, and thus causes the corresponding floating point trap, which is delivered to the kernel as a `#XF` exception. The kernel will translate this into a `SIGFPE` and deliver it to FPSpy's signal handler. FPSpy will then record the errant instruction execution.

At this point, FPSpy has a problem: it needs the errant instruction to actually be executed. We want to do this without any emulation or even understanding of the size of the instruction. To accomplish this, the handler now masks the floating point traps, and returns. This will cause the instruction to execute to completion. Unfortunately, so will all subsequent instructions, including the floating-point instructions we want to track. To avoid this, just before it returns from the signal handler, FPSpy also switches the CPU into single-step mode ("trap mode"), by setting the `TF` bit on the `%rflags` register. It also sets the state machine to AWAIT_TRAP. The result of this is that the errant instruction executes to completion, and then the CPU raises a trap exception when fetching the next instruction. The kernel maps this to a `SIGTRAP` and delivers it to the thread.

FPSpy's `SIGTRAP` handler notes that the system is in AWAIT_TRAP mode. Getting a `SIGTRAP` in this mode indicates we are done with handling the errant instruction. The handler disables the CPU single-step, zeros the floating point exceptions, and unmasks the floating point traps. On returning from the handler, it sets the state machine to AWAIT_FPE. We are now executing the instruction after the errant one, looking for the next floating point trap.

3 Architectural independence

Designing an appropriate set of abstractions for the hardware features FPSpy depends on is subtle for several reasons:

- The IEEE 754 standard does not mandate how features will be exposed at the architectural level; it does not require the existence of specific registers, for example.
- The IEEE 754 standard does not require that floating point traps be implemented. In the absence of floating point traps, FPSpy should still be able to run, albeit restricted to aggregate mode.

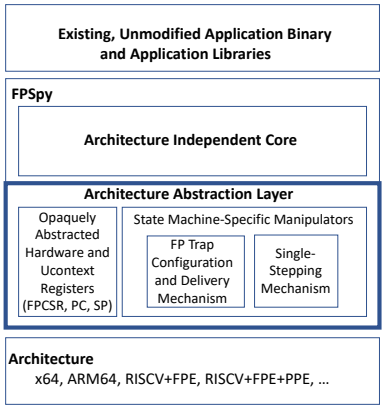

Figure 1: Overview of architectural independence.

- Various architectures may stray from or expand on the IEEE 754 spec. For example, an architecture may not support denormalized numbers at all, support the DAZ/FTZ features mentioned above, or add additional exception types.
- A tool like FPSpy needs to be able manipulate the machine state directly (i.e., read/write specific registers), and indirectly (reading and writing stored state that the kernel will restore, such as a monitored thread's register values in a `ucontext_t` passed to a signal handler.[5]
- While the baseline mode of operation in individual mode is driven by signals, we want to enable faster delivery mechanisms as well, such as specialized kernel support (elaborated on in Section 3.7) or specialized hardware support (Section 4).
- As a tracing tool, FPSpy needs to record machine-specific state, but we want to avoid baking any of this into its recording mechanisms and formats.

Figure 1 gives a high-level view of the abstraction design, which we elaborate upon in this section. Our design was also informed specifically by the x64, ARM64, and RISC-V architectures, and we have implemented the abstractions for those three.

3.1 Interrogation

A small set of functions allow the FPSpy core to determine if the architecture supports floating point traps, DAZ/FTZ, extended floating point exceptions beyond those mandated by the standard. These are aligned with the current needs and features of FPSpy.

3.2 Per-process and per-thread initialization

We assume that some implementations will have to maintain their own internal state for either processes or threads or configure the machine appropriately when these are created. Therefore, we call back to the implementation on a thread or process creation. For our current architectures, this is only really necessary per process, and only for ARM64 and RISC-V, as we describe in Section 3.6.

3.3 Opaquely abstracted registers

All of the architectures we studied provide a user readable and writable floating point control and status register, although the register contents varies considerably. ARM64 is easily the most complex of these with many more options than are needed to implement FPSpy, and even splits the control and status into multiple registers in some cases.

Our abstraction requires the implementer to define a singular *opaque* 64 bit floating point command/status register (FPCSR) type, as well as functions to get and set it from either the actual machine register(s) or from a `ucontext_t`. This allows FPSpy to record this value in a trace without being able to understand it. Architecture-specific post-processing tools can decode its meaning later. Our abstraction has a similar model for the general purpose command/status register (GPCSR). This is included because some architectures use the GPCSR as a portion of the floating point state or to store results. For example, on x64, some floating point comparison instructions actually set `%rflags`, the integer condition code flags, similar to integer comparison instructions.

There are some cases in which the FPCSR, whether the actual machine register, or the one captured in a `ucontext_t`, cannot be entirely opaque to the core of FPSpy. The most important of these has to do with manipulating the rounding mode and DAZ/FTZ status (Section 2.2). To handle this case, we provide a manipulation API for rounding that includes the union of all rounding features we have found on our target architectures. An easier case is in manipulating the machine floating point state so that it is safe to run floating point code within FPSpy itself.

The abstraction also requires that we be able to extract universal registers, specifically the instruction pointer and stack pointer, from a `ucontext_t`. We also require access to a cycle counter.

3.4 State machine-specific manipulation

Because the range of features supported on a typical floating point implementation tends to considerably exceed what is needed for FPSpy, we have designed most of the interface around the specific needs of FPSpy and its execution model.

One function allows FPSpy to inform the implementation of the specific traps that will be enabled in AWAIT_FPE and disabled in AWAIT_TRAP. These are given using the standard `libm` macros. Similarly, the rounding scheme that should hold is set using another function, using the standard macros. Internally, the implementation translates these to masks that will be used on the actual control/status registers, which is where these are typically set.

The abstraction then also includes specific functions where FPSpy can set or reset the previously selected floating point traps, set or reset single-stepping traps for the current instruction, set or reset the previously configured rounding mode, and capture the extant exceptions (translated to standard names).

3.5 x64, ARM64, and RISC-V implementations

We factored the existing x64 implementation according to the abstraction design given above, reusing virtually all of the x64-specific code under the new abstraction layer.

We next implemented the abstraction from scratch for the ARM64 architecture. The ARM64 implementation supports all the same functionality as the x64 implementation. Testing this was not easy, however, as many of the ARM64 processors we have access to

[5]More generally, a `ucontext_t` contains the kernel-specific state of an interrupted thread, plus a pointer to the architecture-specific state, usually called the `mcontext_t`.

simply do not support the floating point traps necessary to use individual mode, the key challenge in FPSpy portability. After exhausting multiple options, we resorted to using Apple Silicon hardware, which does have full support. To test, we ran Ubuntu in virtual machines under UTM.

For RISC-V, we added support for the F, D, and Q floating point extensions. The RISC-V standard explicitly does not support floating point traps, arguing that adding test instructions is a good substitute.[6] Consequently, the "baseline" RISC-V implementation is straightforward, as it only supports aggregate mode. In Section 4, we will describe our extensions to RISC-V to support individual mode and to enhance its performance.

3.6 Per-thread single-stepping challenge

On x64, our implementation still makes use of x64's "trap mode" operation to cleanly execute instructions we are tracing without needing to decode them in any way. Flipping a bit on `%rflags` simply causes a breakpoint trap (`#BP`) before each instruction.

While ARM64 and RISC-V do also have hardware single-step modes, they are not available when running in user-mode, making them useless for FPSpy. As a result, we implement the behavior of trapping on the the next instruction in a considerably more complicated way.

Similar to implementing a breakpoint in a debugger without hardware support, we patch out the next instruction with a breakpoint instruction. We could do this using the `ptrace` infrastructure of the kernel, but this is slow and complex, and would mean that we could not support running under a program that was already being debugged using `ptrace` (e.g., using gdb). Instead, we have per-process initialization visit the entire memory map of the process and mark all executable regions as being writable. This is necessary to allow us to directly introduce the breakpoint instruction wherever it might be needed.

An even uglier problem shows up if the program being tested is multithreaded, because now it is possible for us to be in the middle of processing an instruction for thread 1, insert a breakpoint, and then have thread 2 hit that breakpoint before thread 1 does. This race condition would cause a nasty surprise. We have two options to mitigate this. First, we could do a world stop (pause all threads when we start to handling a faulting floating point instruction on the `SIGFPE`) and then world resume when we finish handling it (i.e., during the `SIGTRAP`). A second option would be to avoid the world stop, but tag the breakpoint instruction as being for a specific thread. Then, if a `SIGTRAP` occurred on the breakpoint instruction for a different thread, we would pause the thread, wait on the breakpoint instruction to be reverted, and only then resume the thread. Essentially, threads only need to synchronize on these introduced/reverted breakpoint instructions. Achieving that is subtle, however. We use the second approach in this paper.

3.7 Trap short-circuiting

In the above, `SIGFPE` drives individual mode, and is delivered using standard Linux signal delivery. Because of signal delivery generality, the number of cycles to go from a `#XF` to the first instruction of the `SIGFPE` handler in FPSpy can be substantial. Additionally, the return from the `SIGFPE` handler also involves the kernel. On our x64 testbed[7], the time from a faulting instruction (one causing a `#XF`) to the first instruction of the `SIGFPE` handler is about 4180 cycles, while the time from the last instruction of the `SIGFPE` handler back to the faulting instruction is about 1800 cycles, for a total overhead of about 5980 cycles. These times include both kernel and hardware times. The hardware time to get the fault into the kernel is about 380 cycles. Hence, the vast majority of the time on this and similar x64 platforms involves the kernel↔user transition of general purpose signals.

Trap short-circuiting is a kernel-level specialization we have developed that delivers the `#XF` into a user-level handler in FPSpy as quickly as possible by replacing the general-purpose Linux signal mechanism with a bespoke design. This is implemented as a kernel module that processes can subscribe to. The kernel module forcibly modifies the `error_entry()` stub in the kernel, which is the entry point for instruction exceptions. It provides a `/dev ioctl()` interface that a cognizant user-level application, like FPSpy, can use to register its own callback function for `#XF`. If this is not done, the modified stub simply uses the standard `math_error()` handler. Otherwise, the stub invokes a function in the kernel module. This function in turn causes the return from exception to vector directly to the user's registered user-level, bypassing the normal signal delivery. The return from exception places enough information on the stack so that the user's handler can figure out what is going on for this specific case. From here, our handler in FPSpy constructs enough of a fake `ucontext_t` so that the remainder of the system is unchanged.

The x64 version of single-stepping (§3.6) is based on being able to place the CPU into trap mode. This makes returning from the handler particularly challenging since once trap mode is set, the very next instruction executed will fault. This might appear to necessitate an expensive return through the kernel, but we have engineered the return logic to execute a same privilege-level transfer instead. This operation hinges on the `iret` instruction, much like it does in the kernel. The `iret` instruction pops an interrupt frame off the stack and simultaneously changes the instruction pointer, the stack pointer, associated code and stack segments and the `%rflags` register (which includes the trap mode bit). Usually, `iret` is used to return from privileged operation (kernel mode) to unprivileged operation (user mode). We instead manufacture an interrupt frame and an `iret` that results in a return from unprivileged operation (user mode) to unprivileged operation (user mode). Since trap mode is a user mode bit on the flags register, the manufactured interrupt

[6] We argue that in a scenario like FPSpy's, where a production binary is being used, this is non-starter. Additionally, if a user is looking for an uncommon event, an Invalid for example, the runtime cost of the test instructions will be born all the time, not just on the rare times that a NaN actually occurs. Additionally, if these "ifs" were introduced at compile-time, they could, arguably, serve as an optimization blocker.

[7] Our x64 testbed consists of a Dell R6515 with a 2.85 GHz AMD EPYC 7443P 24 core/48 thread processor and 512 GB RAM. It is running Ubuntu 22.04.4 LTS (Jammy) with the stock 5.15.0-91-generic Linux kernel. For compilers, the stock gcc 11.4.0 and clang 14.0.0 compilers were used.

Component	LOC
Changes to core	~1000
x64 support	564
ARM64 support	929
RISC-V support (FPE)	260
RISC-V support (PPE)	237
RISC-V support (ESTEP)	63
x64 kernel module	458 (C), 96 (asm)
FPSpy kernel module support	60 (C), 121 (asm)

Figure 2: Scale of changes to FPSpy required to achieve architectural independence with support for three architectures. The total codebase size is about 5000 lines of code (almost entirely C).

frame can include it without a privilege error.[8] Through this technique, we are able to return to the faulting instruction with trap mode on without involving the kernel at all, unlike a `sigreturn`.

Using this approach, which is currently only available on x64, the time (and overhead) from the faulting floating point instruction to the first instruction of handler is reduced from 4180 cycles to 730 cycles (5.7x faster), and the time from the last instruction of the handler to the restart of the faulting instruction is reduced from 1800 cycles to 100 cycles (18x faster), for an overall reduction from 5980 cycles to 830 cycles (7.2x faster). Our kernel module does not include trap short-circuiting for the breakpoint trap (`#BP`), but the performance differences would be just as stark.[9]

When FPSpy is being used to track commonly occurring events, such as (particularly) Inexact, Underflow, or Denorm, trap short-circuiting considerably reduces the application overhead, up to the reductions noted above for the core mechanism. When FPSpy is used to track (hopefully) rare events, such as Invalid or Overflow, the application overhead is already close to nil.

Note that our RISC-V implementation can completely avoid the kernel altogether, which results in an even more dramatic reduction when handling commonly occurring events.

3.8 Scale of changes

The abstraction layer comprises 4 types and 29 functions. Figure 2 illustrates the scale of changes to the previous x64-specific FPSpy implementation, as well as the lines of code needed to implement the abstraction for the x64, ARM64, and several variants of the RISC-V architecture (the FPE and pipelined FPE variants are the subject of later sections of the paper).

4 RISC-V

A background on RISC-V, its implementations, and the Firesim FPGA-accelerated simulator is needed to understand our floating point trap and PPE extensions to RISC-V and their evaluation.

RISC-V is an open-source instruction set architecture (ISA) [32] that is receiving increasing interest from all directions, including high-performance computing. One of RISC-V's key features is that it has customization *built into* the specification, allowing researchers to develop new ideas without creating a whole new specification.

4.1 Exceptions and delegation

RISC-V's privileged specification [34] defines multiple privilege levels and how to handle changing between them. A RISC-V CPU intended to run a classic multi-user operating system, i.e. Linux, will support three privilege modes: Machine (M), Supervisor (S), and User (U). The boot-code and platform-specific firmware will run in Machine-mode while Linux runs in Supervisor-mode, and the user's programs will run in User-mode.

Instruction exceptions and classic CLINT (Core-Local INTerrupts)-based interrupts are defined by the specification to *always* jump to the highest privilege mode (Machine-mode). The code operating at a higher privilege mode can choose to send the event to a lower privilege mode. Delegation is an optimization over this default behavior where the higher-privileged mode specifies to the hardware that the event should be delivered to a lower-privileged mode *directly*. Delegating exceptions and core-local interrupts means redundant privilege jumps and code are never executed, which is critical to high performance on RISC-V cores with multiple privilege levels.

For example, if a user program experienced a page fault, the default behavior would have the core jump to firmware, the firmware would determine that this was a page fault, and then jump to supervisor (the kernel), which again determines that this was a page fault and handle it. If machine-mode opts to delegate page faults, then the processor will immediately jump to supervisor mode (the kernel), completely skipping machine-mode and firmware.

For a time, RISC-V included the N-extension [33], which allowed for delegation of interrupts (not exceptions) all the way to User mode. This has since been removed, although it is not clear why. Our PPE extension is influenced by the N-extension, but is focused solely on instruction exceptions.[10]

4.2 Elements of our implementation

We leverage Chipyard [1] to build complete RISC-V System-on-Chip (SoC) designs, allowing us to quickly iterate on architectural modifications while producing fully-functional SoCs. Chipyard allows the developer to experiment with each part of the SoC in a drag-and-drop fashion, including CPUs and their microarchitecture. Chipyard also provides the Berkeley Hardfloat [31] library to support IEEE-754 floating-point. Our work requires modifications to both Rocket-Core [2] and SonicBOOM [36], but did *not* require modifications to Hardfloat.

Rocket-Core and BOOM both use the Berkeley Hardfloat [31] library to provide a suitable IEEE-754 floating point implementation. As a library, Hardfloat sets the floating-point event flags as required by IEEE-754, but the design using Hardfloat is responsible for trapping. In the core, the floating-point unit (FPU) is connected

[8] Note that this is not an vulnerability. The `iret` will fault if the interrupt frame changes parts of the `%rflags` register that are not accessible in the initial mode, or if an attempt is made to load a segment with higher privilege than the initial mode.

[9] A subtlety in implementing trap short circuiting in the way we do, which works on an out of the box, unmodified kernel binary, is that the kernel module needs to surgically live-patch the normal entry point for the trap/exception of interest. This is not in principle difficult to do, but in practice requires a deep dive into the core kernel to find that entry point.

[10] Intel Sapphire Rapids and later microarchitectures support the UIPI extension, which allows interprocessor interrupts (IPIs) to be initiated and terminate directly from/to user space. To the best of our knowledge this feature does not extend to external interrupts or exceptions. The recent xUI proposal [3] proposes to extend UIPI to handle external interrupts and other capabilities needed for user-level networking and evaluates a design in Gem5 simulation. This proposal does not include instruction exceptions. A PPE extension for x64 would allow us to improve further on the trap short circuiting kernel module introduced in §3.7.

to the core's CSRs, so that floating-point events can be detected by the wider core and software.

The RISC-V floating-point extensions (F-, D-, and Q-extensions) all define that exceptional floating-point events, such as a NaN or rounding, do not trap. Instead, software is expected to **manually** check the floating-point Control and Status Registers (CSRs) to determine if an exceptional event occurred. Our understanding of the design is that execute stages and their functional units are expected to never produce exceptions. This gives the potential for very simple as well as very high performance implementations.

Our implementation is two-fold: already-existing CSRs need to be taught that floating-point exceptions can happen and the core's microarchitecture needs to raise this exception when running. Floating-point events trigger an exception when a floating-point instruction changes the flags and matches the enable mask.

Rocket-Chip and Rocket-Core: Rocket-Chip [2] is a complete tile-based SoC with a Rocket-Core at its heart. Rocket-Core is a single-issue five-stage in-order pipelined implementation of RISC-V GCB ISA and RISC-V's privileged architecture, supporting all four privilege modes (M, H (Hypervisor), S, and U). We modified Rocket-Core's CSR module to create the interface necessary for floating point traps and PPEs. This and several other ancillary components were later reused in our SonicBOOM implementation.

We initially built a version of floating point traps and PPEs for Rocket-Core but eventually abandoned it due to incompatibilities with Rocket-Core's pipeline design. The FPU is pipelined *in parallel* with the main core, with the effects of a floating-point instruction only committing to the core's global state in the main core's write-back stage. As a result, an integer or load/store instruction that follows a trapping floating point instruction is effectively racing with the newly introduced exception. Consequently, it is possible for the integer or memory instruction to complete, even though it should not have due to the exception in the previous floating point instruction.

A solution is to have the FPU raise the exception in the execute stage, but this means the FPU must entirely block the main core's pipeline. This would prevent all instructions from flowing further down the pipeline until the FPU completes. Rocket-Core is not designed for that to be done easily. This issue does *not* mean floating-point traps cannot be implemented in Rocket-Core, but that the core would need to be heavily re-architected to support this feature. Instead, we rebased our work on the SonicBOOM out-of-order core, since such a core must already have the machinery required to handle variable-duration instructions and rolling-back already-executed but uncommitted instructions.

We also modified Rocket-Core's CSR module to implement PPEs by extending the delegation model to allow delegation of exceptions directly to user-space. Delegation can be configured only from S-mode, which means the kernel can select which exceptions it will allow to be delegated to user mode. The U-mode code can only request delegations from the kernel. Note that a floating point trap is trivially delegatable since a typical kernel (e.g., Linux) itself contains no floating point code, and already simply turns the trap into a signal injection to the user process.

SonicBOOM: SonicBOOM [36] (often shortened to just BOOM) is a high-performance micro-coded Out-Of-Order (OOO) RISC-V CPU that plugs into Chipyard's SoC framework; allowing the hardware designer to "drag-and-drop" an out-of-order core into already-existing designs. Unlike Rocket-Core, BOOM cannot be used stand alone; it relies on Rocket-Chip to provide key components, such as supported instructions, CSR management, privilege switching, caches, TLBs, and more. BOOM decomposes each RISC-V instruction into one or more micro-ops (μops) and uses those to complete the provided instruction. BOOM, like every out-of-order core, uses a Re-Order Buffer (ROB) to allow in-flight micro-ops to be executed out-of-order and their results made visible in-order. This *centralized management of μops* makes handling data or exception dependencies between variable-duration instructions trivial.

The majority of our changes involve teaching the Re-Order Buffer (ROB) that floating-point instructions are also now a source of traps. First, each μop maintains a copy of the floating-point trap enable mask at the time it is dispatched to the ROB. After the μop completes its execution on the FPU, if the result has altered its FP flags in a way that matches the mask, the μop is marked as exceptional in the ROB. When the μop reaches the head of the ROB and is about to be committed, the ROB raises the fact an exception occurred, which passes information to the CSR file, causing a trap. Since the ROB commits instructions in program order, the fact that the trapping floating point instruction races with subsequent instructions becomes a non-issue.

Our PPE implementation was trivially rebaseable onto BOOM, since all of the logic for PPEs is contained in the CSR file shared with Rocket in any case. The only micro-architecture-specific change we made was to teach BOOM how to decode two new instructions into μops: The `URET` instruction returns from a PPE and is nearly identical to existing system-return instructions, while the `ESTEP` instruction is a variant of the existing `EBREAK` breakpoint instruction that delivers its exception via PPE.

4.3 Verilator, FPGAs, and FireSim

We debugged our designs largely using the pure software, cycle-accurate Verilator simulation framework [30]. This has the advantage of presenting a copious amount of data down to the wire-level, but it is extremely slow and so instruction sequences must be carefully constructed for testing.

For the evaluation results in this paper, we employ the widely-used and well-validated FireSim tool [18], which is a framework for FPGA-accelerated high-speed cycle-accurate simulation. Performance critical parts of the design are put through the Vivado synthesis toolchain, which targets an AMD Virtex UltraScale+ VCU118 Evaluation Kit [35], which has an XCVU9P-LGA2104E FPGA at its heart, with the FPGA fabric targeting a 90 MHz clock frequency. We used the most recent tagged release of Firesim, 1.20.1, with minor updates to synthesis-related components. This version of Firesim uses GCC 13.2.0 and Buildroot Linux 6.6.0. We did **not** modify GCC, instead we used inline assembly to access the additional CSRs and instructions that we added.

While the synthesis process involved in targeting FPGA-accelerated simulation may take hours to complete, the end result is a cycle-accurate simulated system that operates at real-time speeds sufficient for human interaction. Importantly, the RISC-V processor, including its DDR controllers are implemented directly on the FPGA fabric, and tie to the DRAM surrounding the FPGA on the board. Uncore components were allowed to be simulated at different clock

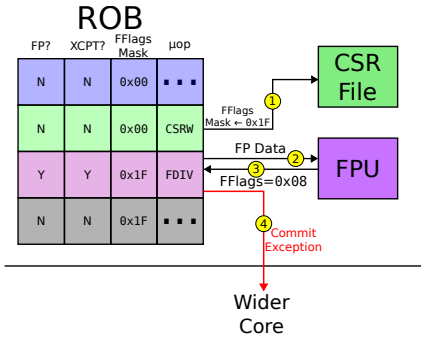

Figure 3: Changes to BOOM needed to support FP Traps.

frequencies, and aspects of the design irrelevant to understanding floating point trap and PPE performance (e.g., disks, UARTs, NICs) are implemented in software. Consequently, our design is fast enough to quickly boot a normal Linux distribution (e.g. Fedora), allowing the user to simply SSH into the simulated machine. This full featured Linux environment is then sufficient for executing complete real program binaries directly and under FPSpy.

5 Floating point traps on RISC-V

It was relatively straightforward to add floating point traps to BOOM, but the devil was in the details, requiring extensive debugging to deal with corner cases that can occur due to commit logic and branch predictor assumptions. Additionally, because RISC-V does not natively include floating point traps, a range of modifications to the firmware and the core Linux kernel are needed. Interestingly, such changes are simpler in the case of PPE delivery.

Hardware: Berkeley Hardfloat, on which BOOM's floating point is based, already computes all the necessary source and result attribute information needed to achieve our goals. Indeed, these attributes already feed the `fcsr` condition code bits.

The majority of our work of implementing floating-point traps was done in Rocket-Chip's CSR file, which implements all the CSRs from the specification, and manages control- and privilege-flow changes because of exceptional events. A new trap cause was created in the custom space, ensuring no existing software could accidentally produce our new floating-point traps. Just one new CSR was added to the CSR file to track the floating-point trap enable mask.[11] We then enable floating-point traps delegation as normal.

Figure 3 shows the hardware changes to the cores required to support floating-point traps. Software can update our added trap enable mask CSR with (1), which propagates to later µops. From this point, when any floating-point µop is sent to the FPU for execution (2), its completion will return both the result of the computation and the µop's `FFlags` result (3), which will be stored in the ROB. If the returned `FFlags` matches the previously set trap enable mask CSR, the ROB will raise an exception to the surrounding core (4), making it visible to the rest of the core and quashing the instruction and dependent instructions. We introduce a new exception number to disambiguate it.

[11]We considered placing the trap enable mask on the existing `fcsr`, as is effectively done on ARM and x64; there is plenty of reserved space on `fcsr` to do this. In the end, we opted instead to add a new CSR to maximize orthogonality with other architectural extensions that may touch `fcsr`.

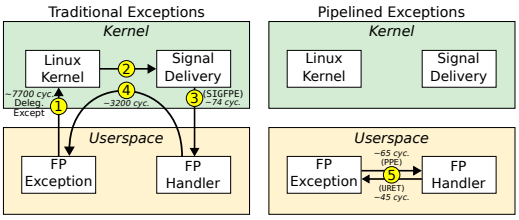

Figure 4: Changes to kernel dispatch to support FP Traps via signals and PPEs. PPEs completely bypass the kernel, effectively jumping to a pre-registered userspace handler.

Which-ware	Component	LoC
Hardware	Hardfloat	0
	FP Traps	BOOM: 150, Rocket: 20
	PPEs	BOOM: 1, Rocket: 150
	FP Traps via PPEs	BOOM: 1, Rocket: 5
Software	OpenSBI Firmware	10
	Linux kernel	200
	PPE kernel module	450

Figure 5: Scale of changes to hardware float system, RISC-V, OpenSBI, and Linux kernel to enable floating point traps and pipelined precise exceptions. Hardware code is in Chisel. Software is in C.

Software: OpenSBI, and Linux: To support the novel floating point trap, OpenSBI (the firmware) needed to be extended to to always delegate floating-point traps up to the next privilege mode (S-mode, or the kernel).

The core Linux kernel also had to be extended to understand the new trap code, and a trap handler had to be written for it and registered with the kernel. By default, our trap handler uses the traditional, existing `SIGFPE` interface to deliver floating-point traps to userspace, mirroring other architectures. The kernel's trap entry code and internal floating point barriers had to be modified to manage the newly introduced trap enable mask CSR correctly. Finally, context-switching code was also extended to properly manage the trap enable mask CSR on a per-thread basis.

The left hand side of Figure 4 illustrates how a floating point trap due to an instruction in a user program flows via hardware to the kernel (1), which (2) translates it into a signal injection of `SIGFPE` (with its architecture-independent semantics) (3), which lands in user's installed signal handler. Eventually, the handler executes a signal return (4), which clears the signal in the kernel and returns the process to normal execution. The right hand side of the figure illustrates how traps are delivered when PPEs are used — the kernel is simply not involved, except in managing the enable mask CSR.

Scale of changes: Figure 5 shows the total scale of all changes to the Berkeley Hardfloat system, the RISC-V cores, the OpenSBI firmware, and the Linux kernel needed to add FP traps and make them deliverable by either signals or PPEs.

6 Precise Pipelined Exceptions (PPEs)

Precise Pipelined Exceptions (PPEs) allow some traps to be delivered *directly* to user-mode, i.e. user applications, instead of delivering the trap to the kernel, and the kernel signaling the user process. PPEs were a considerably more complex addition to the RISC-V core than

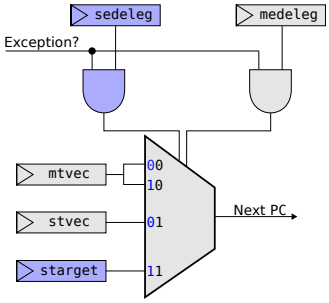

Figure 6: Changes to RISC-V core needed to support PPEs. Each of the new CSRs are registered and the multiplexor gained just one input, none of which affect the critical path.

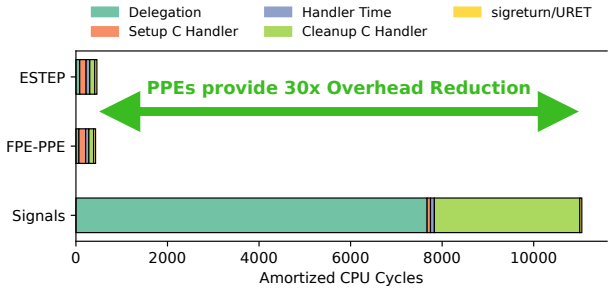

Figure 7: PPEs drastically reduce the cost of handling an exception by removing entrances to the kernel, both when the exception is first found and when re-entering the kernel to clear the signal.

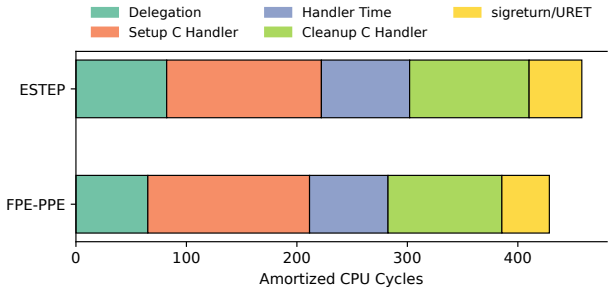

Figure 8: Breaking down the costs of a FPE-PPE and ESTEP-PPE. A majority of the time is spent setting up for and cleaning up after C because we save all integer registers and some CSRs to the stack.

floating point traps on BOOM, but required considerably simpler software support because the kernel can be largely uninvolved.

Hardware: PPEs are built on top of RISC-V's existing trap delegation. We introduce five new CSRs and one instruction. The new CSRs mirror the existing delegation CSRs and comprise a cause register, delegation mask registers, and the exceptional-PC address. The new URET instruction also mirrors the MRET and SRET instructions. This expanded family of system returns read and write several CSRs atomically and *change* the privilege mode, which is different than the normal RET instruction. URET specifically provides the capability to return from the user-level handler back to user code (i.e., same privilege user→user return).

Delivering traps directly to user-mode means there is no context switch out of the user-process because the process has explicitly opted into handling these traps, (5) in Figure 4. This has a variety of benefits, two of the largest are that the usual hardware need to flush the entire pipeline for security can be omitted (we are simply entering a different part of the user program), and various caches do not need to be flushed (there is no possibility of a side channel between actors since there is only one actor). This means an exceptional control flow event effectively turns into an unconditional jump, reducing stalls and improving overall performance.

Critical path unchanged: Figure 6 illustrates the central logic involved in dispatching a PPE. In effect the multiplexer that selects the next PC is widened, which does not affect critical path in our synthesis of the modified core. Similarly, the implementation of URET simply moves values between registers with some multiplexers to choose what register is the target of these movements.

Floating point traps via PPE: Floating-point traps and precise pipelined exceptions are orthogonal ideas, so supporting PPE-based floating-point traps is quite simple to implement. On the hardware-side, enabling floating-point traps to use PPEs amounts to allowing the hardware to delegate FP traps to user-mode. This is a relatively small enhancement; only a handful of lines of Chisel achieved this.

Breakpoint traps via PPE: We further extend our RISC-V platform with the ESTEP instruction, which has behavior *identical* to the normal EBREAK breakpoint instruction, except that the breakpoint exception is delivered via a PPE and it has a distinct exception code. The distinct exception code results in it having no interaction with `ptrace()` and other mechanisms reliant on the standard breakpoint exception. The hardware-software codesign of floating point traps via PPE, and ESTEP make it possible for a tool like FPSpy to bypass the kernel entirely, other than for setup.

Software: Linux, Module, and Userland Stub: To make the PPE changes to hardware visible to software required no core kernel changes. Instead, we created a kernel module that exposes a character device in `/dev`. The kernel module allows user processes to "subscribe" to PPEs by providing a delegation mask and a target handler address via an `ioctl()`. The one requirement is that the handler function must end with a URET instruction. Until this happens, same-event PPEs are masked.

Since the PPE appears to be a surprise jump, creating a handler can be a bit subtle. To address this challenge, we provide a default assembly stub that allows the user to write PPE handlers in C much like traditional signal handlers. The stub saves all integer registers to the stack, and copies some values from PPE CSRs into the the integer registers. This makes it easy to construct a fake `ucontext_t` which allows a standard signal handler function to be invoked. In the long-term, it would make sense to have this stub be generated by the compiler so that it saves and restores minimal state.

Performance: PPEs reduce the overhead for dispatching to that particular trap's user-level handler by 30× in comparison to the use of signals. Figure 7 compares the costs of signal-based delivery with PPE-based delivery. Figure 8 then focuses on PPE-based delivery to make the cost breakdown easier to see. Here, we use a null handler that is written in C. The constituent costs of delegation, the userland stub, and the URET are on par with the cost of the null handler.

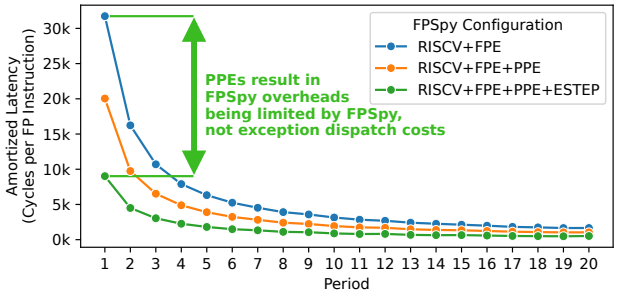

Figure 9: Signals vs. PPE Performance Sweep for FPSpy. PPEs drive the limit of FPSpy overheads down by a factor of three, with the overhead dominated by FPSpy's handling of the exception, not the exception's dispatch.

We expect that further optimization could effectively convert this into a jump, almost completely eliminating the overhead of trap-handling setup. This is particularly apparent in the case of floating-point traps, as the Linux kernel does not use floating-point whatsoever,[12] so the only source of floating-point traps *must* come from user programs.

7 FPSpy on augmented RISC-V

We compare three versions of our RISC-V machine: *RISCV+FPE*, *RISCV+FPE+PPE*, and *RISCV+FPE+PPE+ESTEP*. When running on RISCV+FPE, floating point traps are detected in hardware and delivered by the kernel via `SIGFPE`, while breakpoints are injected using the `ebreak` instruction, which faults to the kernel which in turn delivers a `SIGTRAP`. In RISCV+FPE+PPE, floating point traps are delivered by PPE, and breakpoints can be delivered as before. In RISCV+FPE+PPE+ESTEP, both floating point traps and breakpoints are delivered using PPEs and the kernel is uninvolved.

The FPSpy implementation for RISC-V for all three versions is described in §3.5, with a breakdown of the implementation given in Figure 2. The implementation is slightly bigger than the ARM to provide support for PPEs, and both are about twice as large as the x64 implementation due to the unavailability of a single-step mode.

Functional completeness: On all three versions, FPSpy passes its own acceptance tests for individual mode and is able to run successfully in that mode under the NAS 3.0 benchmark suite [4]. This brings it to feature parity with the x64 and ARM platforms.

Performance: The raw numbers shown in Figures 7 and 8 make it clear that the baseline costs of handling a floating point trap or a breakpoint trap via PPEs improve on handling them by signals by a factor of 30 or more. Of course, FPSpy additionally decodes the circumstances of these events and captures this information in a trace. FPSpy processing is about 9000 cycles per event, with the event delivery costs added to this. With signal-based delivery, event delivery (2× ~12,000 cycles) dominates the overhead, while with PPEs this drops to less than 1000 cycles and thus FPSpy processing dominates the overhead.

The greater the rate of FP traps, the greater the benefit FPSpy will see from PPEs. If, for example, the intent is to monitor for NaNs using FPSpy, then there is little difference in the overhead between the three versions in the expected case where NaNs are rare. On the other hand, if the intent is to observe rounding behavior, FPSpy will see maximum benefit from PPEs because rounding events are common in floating point arithmetic. Figure 9 shows effect on overhead for PPE versus signals as a function of the incidence of traps, which we sweep. Period k here means that every kth floating point instruction causes a trap. In the limit, PPEs reduce the overhead of FPSpy by a factor of three.

[12] In fact, Linux goes so far as to **disable** floating-point support while in the kernel (see `arch/riscv/kernel/head.S` and `arch/riscv/kernel/entry.S`).

8 Related work

Interest in correctness in scientific computing has been growing for years, and this has recently included a workshop [14] to advise the DOE and NSF that resulted in the CS2 program. Floating point arithmetic is a specific area of interest here, and there is some evidence that even with correct numerical methods, developers may harbor confusions about implementation in floating point [10, 12] and compiler optimization might inflict damage [29].

There is a long history of work on tools to improve source code that employ floating point arithmetic (e.g., [5–9, 13, 19–22, 24, 25, 27, 28]. The FPSpy system [11] we extend in this paper aligns with such work, and was the first with the goal of leveraging the floating point traps defined in the IEEE 754 standard and available on x64 processors to operate under unmodified binaries. Floating point trap support is optional on ARM, and as far as we are aware, no FPSpy-like tool currently exists for ARM processors that support such traps. Because RISC-V is explicitly defined not to include floating point traps, trap-driven tools like FPSpy were impossible to build prior to the work we describe here.

A case was made for user-level interrupts as far back as 2002 [26]. The "N" extension to RISC-V [33] defined this concept, but then was rescinded for unclear reasons and never implemented. Intel introduced user-level interprocessor interrupts (UIPIs) with the Sapphire Rapids microarchitecture [17, Chapter 7] and it exhibits good performance [23]. Very recent work by Aydogmus et al [3] shows how to extend UIPI to handle external interrupts. As far as we are aware, however, there is no prior work on user-level delivery of instruction exceptions, as in the PPE component of our work.

9 Conclusions

We have described our experience adding architecture-independence to an x64-specific tool for hardware-based observation of floating point events, and with adding low-overhead support for such observation to an architecture which does not include it. Both the tool and the architecture extensions will be publicly available for x64, ARM, RISC-V (partial functionality), and our augmented RISC-V (full functionality with low overhead). Although our prototype is implemented on top of FPGA-accelerated simulation, our changes could be synthesized as part of a BOOM-based chip design, or incorporated into an alternative RISC-V implementation. Our next steps are to go beyond tracing floating point events. One direction is extending our hardware to support the manipulation of floating point instruction execution. A second is to track flows of floating point values through memory and registers. Our key hardware mechanism for achieving low overhead, the PPE, has many potential uses beyond floating point arithmetic, which we are also exploring.

References

[1] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. https://doi.org/10.1109/MM.2020.2996616

[2] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. University of California at Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[3] Berk Aydogmus, Linsong Guo, Danial Zuberi, Tal Garfinkel, Dean Tullsen, Amy Ousterhout, and Kazem Taram. 2025. Extended User Interrupts (xUI): Fast and Flexible Notification without Polling. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2025)*. 373–389.

[4] 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.

[5] 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)*.

[6] 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)*.

[7] 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)*.

[8] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous Floating-point Mixed-precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 300–315.

[9] Clement Courbet. 2021. NSan: A Floating-Point Numerical Sanitizer. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC)*.

[10] Peter Dinda and Alex Bernat. 2021. *Comparing the Understanding of IEEE Floating Point Between Scientific and Non-scientific Users.* Technical Report NWU-CS-2021-07. Department of Computer Science, Northwestern University.

[11] Peter Dinda, Alex Bernat, and Conor Hetland. 2020. Spying on the Floating Point Behavior of Existing, Unmodified Scientific Applications. In *Proceedings of the 29th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2020)*. Best Paper.

[12] Peter Dinda and Conor Hetland. 2018. Do Developers Understand IEEE Floating Point?. In *Proceedings of the 32rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018)*.

[13] François Févotte and Bruno Lathuilière. 2016. VERROU: Assessing Floating Point Accuracy Without Recompiling. working paper or preprint.

[14] Maya Gokhale, Ganesh Gopalakrishnan, Jackson Mayo, Santosh Nagarakatte, Cindy Rubio-Gonzalez, and Stephen Siegel. 2023. Report of the DOE/NSF Workshop on Correctness in Scientific Computing.

[15] IEEE Floating Point Working Group. 1985. IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985* (1985).

[16] IEEE Floating Point Working Group. 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70.

[17] Intel Corporation. 2023. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3A: System Programming Guide, Part 1.*

[18] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2018. FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 29–42. https://doi.org/10.1109/ISCA.2018.00014

[19] 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.

[20] Ignacio Laguna and Ganesh Gopalakrishnan. 2022. Finding inputs that trigger floating-point exceptions in GPUs via bayesian optimization. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC 2022)*. 14.

[21] Michael O. Lam, Jeffrey K. Hollingsworth, and G.W. Stewart. 2013. Dynamic floating-point cancellation detection. *Parallel Comput.* 39, 3 (2013), 146–155.

[22] 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)*.

[23] Sohit Mehta. 2021. User Interrupts: A Faster Way to Signal. In *Proceedings of the Linux Plumbers Conference (LBC 2021)*.

[24] 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.

[25] 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)*.

[26] Mike Parker. 2002. A case for user-level interrupts. *SIGARCH Computer Architecture News* 30, 3 (June 2002), 17–18.

[27] 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)*.

[28] 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)*.

[29] 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.

[30] Wilson Snyder, Paul Wasson, Duane Galbi, and et al. 2024. Verilator. https://github.com/verilator/verilator original-date: 2019-06-13T22:38:59Z.

[31] ucb-bar. 2024. *Berkeley-Hardfloat.* UCB-BAR. https://github.com/ucb-bar/berkeley-hardfloat

[32] Andrew Waterman and Krste Asanović (Eds.). 2016. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA.* RISC-V Foundation. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[33] Andrew Waterman and Krste Asanović (Eds.). 2016. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA.* RISC-V Foundation.

[34] Andrew Waterman and Krste Asanović (Eds.). 2016. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture.* RISC-V Foundation.

[35] AMD Xilinx. [n.d.]. AMD Virtex UltraScale+ FPGA VCU118 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/vcu118.html

[36] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. (May 2020).