Virtualization So Light, it Floats! Accelerating Floating Point Virtualization

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

Nadharm Dhiantravan
nadharm@u.northwestern.edu
Northwestern University
Evanston, Illlinois, USA

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

Abstract

Floating point virtualization enables unmodified application binaries to utilize alternative arithmetic systems such as MPFR without code changes, but its performance overhead is a barrier to adoption. The existing trap-and-emulate model suffers from a significant virtualization bottleneck using general-purpose signal delivery mechanisms which take thousands of cycles. We introduce three techniques to reduce virtualization overhead. Trap short-circuiting bypasses general-purpose signal delivery for an 8x reduction in trap delegation overhead. Instruction sequence emulation amortizes trap costs by emulating multiple instructions per trap, achieving up to 32x reduction in trap frequency. Finally, kernel-bypass for correctness instrumentation eliminates traps and signals for correctness and reduces related overheads substantially. Our implementation within the FPVM system on x64/Linux demonstrates a 10x reduction in per-instruction overhead which, compared to the lower bound performance set by the alternative arithmetic system, drops virtualization overhead from up to 20x to 1.65x. This is for the alternative arithmetic system that is the worst case for virtualization overheads. More expensive systems, like MPFR, fare even better.

CCS Concepts

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

Keywords

virtualization, floating point arithmetic, software development, IEEE 754

ACM Reference Format:
Nick Wanninger, Nadharm Dhiantravan, and Peter Dinda. 2025. Virtualization So Light, it Floats! Accelerating Floating Point Virtualization. In *The 34th International Symposium on High-Performance Paralleland Distributed Computing (HPDC '25), July 20–23, 2025, Notre Dame, IN, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3731545.3731584

1 Introduction

Motivation. Floating point virtualization enables unmodified application binaries to execute using alternative arithmetic systems, such as high-precision libraries (e.g., MPFR) or unconventional models (e.g., posits, interval arithmetic, and rational arithmetic). Alternative arithmetic systems permit higher precision, better error bounds, or other features not available in standard IEEE 754 floating point arithmetic. This facilitates assessment and experimentation of alternative arithmetic systems, numerical stability, precision, and correctness. Floating point virtualization bridges the gap between the IEEE 754 arithmetic that hardware supports and compilers target, and the alternative arithmetic systems we might want to try out in situ in scientific applications.

The specific floating point virtualization tool we consider in this paper, FPVM [15], implements a "trap-and-emulate" approach, where the hardware exceptions trigger software handling of floating point instructions. The original IEEE instructions in the program run directly on hardware at full speed whenever they can provide a precise result. When an instruction cannot provide a precise result, the instruction is trapped by the hardware, and the trap is delivered to the virtualization system, which then emulates the instruction using the alternative arithmetic system.

While this enables seamless integration of alternative arithmetic, the performance overhead of trap-based virtualization is substantial. Ideally, switching to an alternative arithmetic system would only incur the cost of the alternative arithmetic itself. On modern x64/Linux systems, the cost of delivering a floating point trap to user space via standard POSIX signals exceeds 5,000 cycles per instruction, making trap handling the dominant bottleneck. For many applications, this overhead results in slowdowns exceeding 1000x, limiting where floating point virtualization can be deployed. Our work focuses on eliminating these inefficiencies, ensuring that the overhead of floating point virtualization is dominated by the cost of the alternative arithmetic system itself, rather than the virtualization mechanism.

Limitation of state-of-the-art approaches. Prior work on floating point virtualization has primarily relied on trap-and-emulate mechanisms, where floating point instructions execute natively unless they require intervention (e.g., due to rounding, underflow, or alternative arithmetic). When intervention is needed, the hardware raises a floating point exception, which is delivered to a user-space handler via POSIX signals (e.g., `SIGFPE`, `SIGTRAP`). This approach, exemplified by FPVM, allows applications to seamlessly use alternative arithmetic without modification.

However, this method suffers from several fundamental limitations: The cost of handling a single floating point trap is dominated by kernel-to-user signal delivery, which takes 3,800-5,600 cycles per trap on modern x64/Linux systems. Since only one instruction is handled per trap, this overhead is incurred repeatedly, making the virtualization cost excessively high. Many scientific applications execute tight numerical loops, where sequences of floating point instructions execute in rapid succession. In traditional trap-and-emulate, each instruction in the loop incurs its own trap, exacerbating the overhead. Additionally, due to x64 architecture constraints, floating point values can flow into integer registers and memory locations where traps do not occur naturally. To address this, existing systems insert special traps before instructions that might attempt such operations. These invoke the virtualization system to allow it to maintain correctness, but at the cost of even more expensive traps to invoke it.

These limitations result in extreme slowdowns—sometimes exceeding 1000x—restricting the scenarios in which floating point virtualization is practical to deploy. Our work directly addresses these bottlenecks by redesigning trap delivery, amortizing trap costs, and eliminating unnecessary kernel transitions, significantly improving the performance of floating point virtualization.

Key insights and contributions. FPVM (and other forms of floating point virtualization) has the goal of having its overhead be limited by the performance of the alternative arithmetic system, not the mechanism of floating point virtualization itself. Unfortunately, the initial implementation was far from this goal except for very expensive alternative arithmetic systems. An overhead analysis showed that the key bottlenecks involved the costs of delivering the hardware traps and exceptions to FPVM, and that only a single instruction was handled per trap. In this paper, we describe three software techniques that we have developed to considerably accelerate floating point virtualization, and which we have implemented and evaluated within the context of FPVM. Our implementation is for x64 machines running Linux, but could be adopted to other architectures as well.

To address trap delivery overheads, we present trap short-circuiting. This approach delivers the hardware-initiated traps to the virtualization system as quickly as possible by replacing the general-purpose Linux signal mechanism with a bespoke design for FPVM applications. This is implemented as a kernel module that processes can opt into, and it reduces the kernel-to-user transition cost by 8x.

Instruction sequence emulation improves the implementation of trap-and-emulate by emulating multiple instructions per trap (and thus amortizing the costs of the transition to the virtualization software). We observe that many applications and benchmarks exhibit runs of IEEE floating point instructions. This should come as no surprise—one would expect that an inner loop, for example, would typically have more than one floating point instruction outside of the simplest of numeric algorithms or lack of optimization. It is not immediately obvious, however, that a run of floating point instructions should *all* be emulated. After all, if we emulate an instruction that would otherwise have been run directly by the hardware, emulation for that instruction will necessarily be slower, regardless of the benefits for the sequence it is embedded in.

Kernel bypass for correctness instrumentation improves the costs of correctness traps noted earlier by turning them into special calls between the instrumented program and FPVM, avoiding the kernel altogether. Additionally, we have considerably improved the rate at which correctness traps occur by replacing FPVM's static analysis and patching approach with a profiling and patching approach.

We evaluate trap-short circuiting, instruction sequence emulation, and kernel bypass for correctness instrumentation on x64 processors via an implementation of the techniques within a Linux kernel module, and by enhancement of the open FPVM codebase. We then employ this system on a range of applications and benchmarks, measuring and analyzing the improvements on overhead.

Our contributions are as follows:

- We summarize the performance issues that arise in a trap-and-emulate approach to floating point virtualization. These primarily stem from use of hardware traps and exceptions to drive virtualization. On current x64 systems, these are expensive for the hardware to deliver to the kernel (hardware → kernel), and expensive for the kernel to deliver to user-space (kernel → user) using standard POSIX signals (§2).
- We describe the design and implementation of trap short-circuiting, which speeds trap delegation and return by 8x (§3).
- We describe the design and implementation of sequence emulation, which amortizes trap delegation costs over multiple instructions. The amount of amortization depends on the application, but range from 2 to 32 across our tests. We include an extensive discussion and workload characterization for this aspect of the design (§4).
- We describe the design and implementation of kernel bypass for correctness instrumentation, which avoids both the hardware → kernel and kernel → user costs when FPVM must be invoked due to limitations of the x64 hardware and static analysis. This technique, combined with a profiling based approach to place correctness traps practically eliminates the overhead of correctness from the original paper (§5).
- We consider the average cost per emulated instruction across a range of benchmarks and applications, using the lowest cost alternative arithmetic system in order to emphasize the virtualization costs. We find that our techniques reduce the average cost per emulated instruction by up to 12.5x.

Our techniques move the bottleneck to the emulation core, even for the lowest cost alternative arithmetic system. This moves us much closer to the goal of floating point virtualization limited by the alternative arithmetic system instead of the virtualization mechanisms.

Experimental methodology and artifact availability. FPVM is publicly available from buoyancy-project.org, presciencelab.org or https://github.com/PrescienceLab/fpvm.

Evaluation with worst-case alternative arithmetic system: For almost all of this paper, we evaluate FPVM using the "Boxed IEEE" alternative arithmetic system. This system uses hardware double-precision floats to perform arithmetic, but places them in boxes on the heap that are referenced through NaN-boxed pointers. This system is the fastest alternative arithmetic system and is thus a "worst case" system for FPVM, as the overheads of virtualization become the dominating factor in performance reduction. This allows us to focus on the goal of this paper: reducing floating point virtualization costs.

Our evaluation also includes measurement using the MPFR alternative arithmetic system. Because MPFR is itself more expensive than Boxed IEEE, overheads are much closer to those of MPFR itself.

2 Floating point virtualization

Before diving into the techniques of this paper to accelerate FPVM, we will first discuss floating point virtualization as it stands today. Floating point virtualization seeks to extend the hardware floating point arithmetic environment available to an existing, unmodified application binary without requiring changes to the application binary. The architectural instructions embedded in the binary appear to execute as normal, but instead are selectively emulated using the alternative arithmetic system. Data values also appear to flow through the program as normal, although they are selectively represented using the alternative arithmetic system. Floating point virtualization thus enables scientific applications, even in "blessed" final binary form suitable for production environments, to use alternative arithmetic systems, such as MPFR, without changes.[1] Ideally, floating point virtualization would work equivalently to an OS-level virtual machine monitor, transparently adding this new capability with minimal to no performance consequences outside of the cost of the alternative arithmetic system itself.

Our exemplar for floating point virtualization is our previously published and publicly available FPVM system [15]. We now use its design to capture the salient issues and provide the basic theory of operation.

2.1 User-level virtualization

Despite the goal of virtualizing the hardware, FPVM makes the deliberate design decision to operate at user-level instead of in the kernel. One reason for this is that porting an alternative arithmetic system (MPFR, for example) into the kernel is a daunting challenge, even for kernel developers. Additionally, typical architectures (e.g., x64, ARM, RISC-V) do not privilege floating point state and instructions, and FPVM itself does not need to provide any protection capabilities either. Both of these points mean being in the kernel is not required. Another benefit of being in the user space is that introspection into the running process is much easier and faster.

The overall structure of FPVM is that of an `LD_PRELOAD` library, meaning that it inserts itself early in the dynamic library link order involved with startup of the process to be virtualized. `ld.so` will use its symbols in preference to those of later shared libraries, for example, `libm`. This enables FPVM to override or modify their behavior. FPVM also exports high-priority constructor functions, which are then invoked extremely early, allowing FPVM to initialize well before the `main()` of the process begins to do its own initialization. FPVM's constructors are subsequently invoked on every `fork()`, allowing the virtualized program to spawn further virtualized subprocesses. The startup of new threads using `pthread` or `clone()` is also intercepted so that FPVM can create an execution context for each thread. By default, FPVM also uses standard POSIX signal configuration (`sigaction`) to arrange to have the `SIGFPE` (floating point error), and `SIGTRAP` (breakpoint) signals delivered to its own handlers. The handlers can reassociate the execution context with the thread producing the signal. Virtualization operates on a per-thread basis, and is explained below.

[1]We do not mean to imply that the composite of the blessed final binary and FPVM would also be blessed, merely that floating point virtualization would allow evaluation of alternative arithmetic systems using the blessed binary.

Alternative arithmetic system interface. FPVM has a well-defined interface to the alternative arithmetic system, which allows different choices to be compiled in. In this work, we use Boxed IEEE for the most part, as well as MPFR.

2.2 NaN-boxing

An important difference between FPVM and traditional virtualization is that FPVM needs to use the alternative arithmetic systems's representation of numbers without changing how the application itself uses them. This means that FPVM must somehow represent these alternative numbers in the space used by the floating point numbers that the original machine instructions produce and consume. FPVM accomplishes this using *NaN-boxing* [10, 35]. When a result is produced using the alternative arithmetic system, a pointer to that result is encoded as the mantissa of a 64 bit signaling NaN. When another floating point instruction (e.g. `addpd`) consumes that value the hardware will trap to FPVM (§2.3), allowing the operation to be performed in the chosen alternative math system.

We refer to the process of producing a NaN-box-encoded value in the alternative arithmetic systems as a *promotion.* A NaN-boxed encoded value can also be converted back to a regular 64-bit IEEE 754 value (thus losing the precision or benefits of the alternative math library). We refer to this as a *demotion.*

Differentiating our NaNs, their NaNs, and alternative NaNs. FPVM must be careful to distinguish NaN values which *it owns* and those it does not. FPVM's NaNs encode pointers to objects FPVM has allocated, and FPVM's allocator tracks these pointers. Given a NaN, we check its bit pattern to see if it could have come from FPVM. If not, we assume it is an application NaN. Canonical form NaNs also are detected quickly by this check. If it does have a compatible bit pattern, we extract the pointer from the NaN and check to see our allocator remembers it.

The mantissa space of an IEEE double is 52 bits. For a NaN, two of these are constrained to encode a nonzero bit and to differentiate between signaling and quiet NaNs. This leaves 50 bits to play with. The probability that a randomly drawn NaN happens to both match our pattern and to encode a pointer our allocator returned is $1 - (1 - 2^{-50})^n$ for n active allocations. Even for one billion active allocations, this less than a one in a million chance.

Even this is applicable to only NaNs that flow into the program (e.g. from an input file.) If a NaN is produced by instruction execution, this triggers FPVM, which will perform the operation using the alternative arithmetic system. This might itself produce an "alternative NaN", which is allocated and NaN-boxed by FPVM and thus a non-problem.

2.3 Floating point traps

When a thread starts, FPVM configures a thread-local context, along with the thread's `mxcsr` register to tell the x64 hardware to produce traps when a floating point instruction causes a floating point exception such as Invalid (NaN consumed or produced), Round (result would be rounded), Overflow (infinity produced), Underflow (Zero produced), and Denorm (denormalized number produced). FPVM manages `mxcsr` so that each instruction that raises a floating point exception causes a trap.

By default, x64 dispatches floating point traps into the Linux kernel using standard exception mechanisms. This has a hardware-dependent cost. On our testbed, this dispatch costs roughly 380 cycles, which is quite fast. We denote this cost *hw*. The Linux kernel then takes this exception and injects it back into FPVM as a `SIGFPE` signal. The signal delivery depends heavily on the kernel and the hardware, and on our testbed this cost is roughly 3800 cycles to deliver the signal, with an included cost to return using a `sigreturn` system call of 1800 cycles. We denote the cost of delivering the exception to userspace as *kern*, and returning as *ret*.

When FPVM is invoked by a `SIGFPE`, there are two different routes it can take. First, the signal may be due to a Round, Underflow, Overflow, or Denorm condition. This means that no exact result is possible within the limits of IEEE 754. Here, temporary copies of the input operands are promoted, a result is computed using the alternative arithmetic system, and that result is then NaN-boxed and written to the output operand. In the second case, the signal is due to an Invalid condition (a NaN). If this happened because an input operand was one of our NaN-boxed values, we promote all other inputs which are not already NaNs, invoke the alternative arithmetic system to compute the result, NaN-box it, and write it to the output operand. It is also possible that the Invalid is due to regular input operands producing a NaN output (for example, 0.0/0.0). This is treated like the first case, with the extension that we can store a canonical NaN to the destination to represent that the result is a "real NaN" instead of one of FPVM's boxed ones.

Although FPVM focuses on increased precision, using doubles as the baseline, it could support decreased precision by having every floating point instruction trap. On x64, this can be readily done by disabling the floating point hardware altogether. This is not currently done.

2.4 Instruction Decoding and Emulation

In processing a `SIGFPE` signal, FPVM incurs four main costs: decode, binding, emulation, and garbage collection. The first thing FPVM must do is decode the instruction which caused the exception to determine which registers to use as input, output, and what operation to perform on them. Decoding an instruction on x64 is non-trivial, and as such FPVM features a *decode cache* which records previously decoded instructions for later use. The cost of the lookup in this cache, which is almost always a hit, is denoted *decache*. This cache hits because scientific applications are comprised mostly of loops, and instructions are re-executed often. If FPVM misses in the decode cache, it decodes the faulting instruction with the Capstone disassembler [1] and creates an FPVM-specific representation (which is ultimately intended to also be architecture independent). The cost of invoking Capstone is denoted *decode*.

Once the instruction is decoded, FPVM *binds* the operands, a cost we denote as *bind*. Binding takes the decoded instruction and computes pointers into the the `ucontext` register state and memory to be used as input and output parameters for emulation. Finally, FPVM emulates the instruction using a custom emulator that leverages the FPVM-specific representation and interfaces with the alternative arithmetic system. This cost is denoted *emul*. Recall that we are using Boxed IEEE, the fastest alternative arithmetic system in this paper in order to spotlight the overheads of the system which are *not emulation*. As such, when we measure *emul*, most of the cost captured is external to the alternative arithmetic system (*altmath*).

2.5 Garbage collection

Another cost which FPVM must pay for each `SIGFPE` is to potentially invoke a *garbage collector*. Because FPVM allocates and encodes pointers to heap objects in (often temporary) NaNs, garbage can readily result, and it must be collected to avoid an explosion of memory usage. For example, if a register containing a NaN-boxed encoded pointer is overwritten each time through a loop, each iteration results in a potentially orphaned value in the alternative arithmetic system which is no longer referenced. An interesting side-effect of how these references are used is that despite being heap objects, they must operate *as if* they were values. This means they must be *immutable*, as multiple registers can contain references to the same object, and changing one register's value through mutation would incorrectly change the other.

FPVM's allocator performs garbage collection of values which no longer have live references via NaN-boxed pointers. This is a highly specialized form of garbage collection since the objects managed by the collector never have pointers themselves, and it is only concerned with finding NaN-boxed values. Despite this, the GC is still a very traditional conservative mark-and-sweep collector, and performs a scan over all *writable* pages of virtual memory, marking all objects found in NaN-boxed references, sweeping (freeing) the unmarked objects when done. Thes cost is denoted *gc*.

2.6 Correctness instrumentation

Unfortunately, x64 floating point hardware is not entirely virtualizable. This is due to the ability to perform non-floating point operations on the same registers used for floating point (e.g., the `xmm`/`ymm`/`zmm` register sets), the ability to move between the floating point and general purpose register sets, and because floating point values can flow into integer contexts via memory (for example, by casting a `double*` to a `long*` and reading the bits directly). To address this, FPVM features a static analysis which can conservatively determine points in a program where a floating point value cannot be interpreted as an integer. Equivalently, it determines the points where a floating point result *could* flow and would not be caught by the hardware when accessed. It then patches the binary at these points using e9patch [16] to include an `int3` instruction. At runtime, this instruction causes a hardware breakpoint exception to be delivered to the kernel which is then delegated to FPVM through a `SIGTRAP` signal. At this point FPVM can determine if a NaN-boxed value *is* flowing in this dynamic instance of the instruction. FPVM can then emulate the instruction or single-step over it after demoting any possible NaN-boxed values in doing so. We denote this cost as *corr*, and some applications feature more of these faults than others. In terms of virtualization costs, there is a complex interplay between what the analysis can find statically, the costs of evaluating the problematic instructions dynamically, and the probability of a NaN-boxed value actually flowing through the instruction.

Finally, for code that is not available for static analysis (e.g. shared libraries), the binary is patched with a similar `int3` before the call, and arguments are demoted in a similar way. This is important for

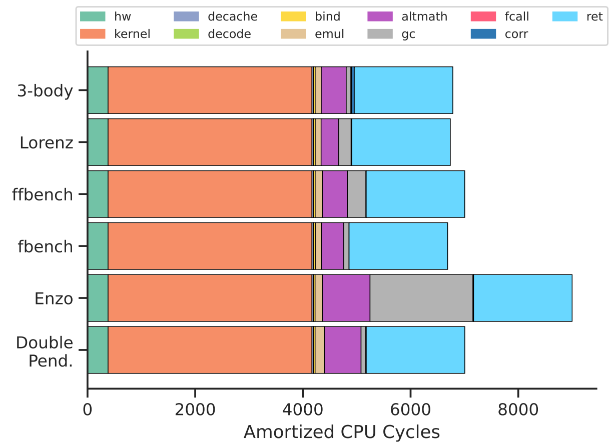

Figure 1: Breakdown of baseline costs (amortized to the cost per emulated instruction) in FPVM using Boxed IEEE as the alternative arithmetic system. The main portions of the overhead come from costs *outside* of emulation (hw, kern, ret).

functions such as `printf`, which perform bit-wise interpretation of floating point values in order to print them. We denote this cost as *fcall*, and the quantity of these traps similarly varies heavily between applications. In a fully virtualizable architecture (described in [15]), the corr and fcall costs would not exist.

2.7 Breaking Down the Costs of Virtualization

Figure 1 shows the baseline starting point of FPVM on our test platform, which includes none of the three optimizations described in this paper. We amortize all costs over emulated instructions to consider the expected cost of any particular emulated instruction. At the core of emulation is the cost of operations in the alternative arithmetic system, which is unavoidable (**altmath**). The other costs are the focus of the paper. To be more specific, in this work we focus on everything other than *altmath* from the figure—hw, kern, decache, decode, bind, gc, fcall, corr, and ret—as described above.[2] Our goal is to reduce the non-intrinsic overheads that are not part of the alternative math implementation to the lowest possible amount—the **altmath** component should dominate each bar of this graph.

Note that, in this paper, we use “Boxed IEEE” as our alternative arithmetic configuration. This just means that we allocate double precision floating point values on the heap and use NaN-boxed references to them, simulating the number of traps from a higher complexity altmath implementation. In the baseline figure, all benchmarks incur the same overhead for hw, kern, and ret, but feature unique overheads for altmath, gc, and correctness, based on their specific implementation characteristics. For example, Lorenz generates less garbage than Enzo as its internal state is much smaller, and 3-body has more correctness traps (*corr*) due

[2] In comparison to the numbers reported in the original FPVM paper, which ran on substantially older hardware and software (Dell R815, 2.1 GHz AMD Opteron 6272, 128 GB RAM, Ubuntu 16.04 with 4.4.0 kernel, gcc 5.4 toolchain), our numbers on current hardware and software are overall comparable, though the new hardware has an hw cost that is about 5x lower in cycles (and this new hardware has a much faster clock).

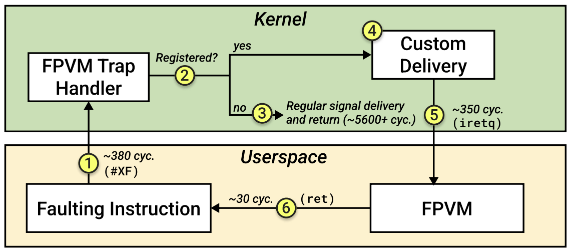

Figure 2: Replacing the SIGFPE signal delivery with trap short-circuiting reduces the overhead of delivery by 8x.

to how it writes more floating point data to the filesystem using `fprintf` (requiring demotions as described above).

3 Trap short-circuiting

Ultimately, FPVM's largest overhead comes from how it is invoked by floating point traps which the kernel delegates using signals. As noted in §2.3, this overhead involves to substantial costs, one being the time for the hardware to dispatch the trap to the kernel (380 cycles), and the other being the time to deliver the `SIGFPE` signal to userspace (3800 cycles). As can be seen in Figure 1, the kernel to user dispatch (*kernel*) is the dominant cost, which was also found to be the case in the original FPVM paper. The signal delivery and return mechanism is entirely in software, and has the potential to change drastically with a different design.

To reduce this overhead to zero, a highly aggressive redesign of FPVM might be to move it to operate *entirely within the kernel*, which would result in $3800/380 = 10x$ lower overhead for *kern+hw* on our platform. Of course, this would be a monumental effort as many of these arithmetic systems are implemented with user-level assumptions, and the kernel is not permitted to use floating point itself—a requirement of some alternative math libraries such as MPFR. Therefore, we present a system to *reduce*, rather than eliminate, the overhead of trap delegation delivery back to userspace to the lowest level permitted by the kernel and the hardware while maintaining compatibility with unmodified Linux on x64 systems.

3.1 Design and implementation

To reduce the overheads of trap delegation, we must first understand the overheads of signal delivery. Normal signal delivery (for example, `SIGFPE`) is a general purpose mechanism and involves fairly complex processing. Signals may be sent to descheduled threads, can be masked, and their priorities can be changed. Additionally, when a signal is delivered, the user must enter the kernel a second time to notify it that further signals can be delivered using `sigreturn` before returning back to the faulting instruction. For the sake of FPVM, these general mechanisms are not required, and an alternate co-designed approach which bypasses them can reduce the overhead considerably.

To this end, we have implemented a Linux kernel module which replaces the x86 trap handler for `#XF` (SIMD floating-point exceptions) and delivers the exception to the FPVM at an extremely early point in the kernel. This operation is outlined in Figure 2. When a floating point instruction causes a rounding event, consumes a NaN, or otherwise faults, the CPU triggers a #XF exception which is sent to the kernel in roughly 380 cycles (1). By default, the linux kernel would call a method `math_error()` which ultimately attempts to deliver a `SIGFPE` signal to the user application through the general purpose signal delivery mechanism. However, with our modifications to the trap handler, a user application can "register" itself through an interface in `/dev`. If an application does not register (2), a trap will be delivered via the normal signal delivery mechanism–staying compatible with the rest of the system (3). If registered, the kernel module bypasses the entire signal delivery infrastructure and instead use a custom delivery mechanism designed specifically for FPVM's needs (4).

This mechanism essentially steals the hardware trap from Linux, modifying it to perform the bare-minimum processing required to hand control over to the FPVM user-space handler. First, it edits the x64 interrupt frame to redirect execution upon the interrupt return (`iret`) as well as bypass the red-zone.[3] Then, it stores the necessary state to allow for the user-space handler to unwind back to the faulting instruction. After this, the kernel module returns to the trap handler, executes the `iret` instruction, and execution resumes at FPVM's user-space handler (5).

FPVM's core user-space handler is wrapped by an entry and exit stub. The entry stub acts as the "landing-pad" to which the kernel jumps. It saves a sufficient amount of state in the format of a `ucontext` to roughly match the interface of a signal handler. This allows FPVM to continue operation as if it has received a `SIGFPE`. The FPVM handler for floating point traps can then manipulate this "fake" `ucontext` as it normally would do. When the handler returns to the exit stub, the stub restores the machine state using the in-memory `ucontext`-like data, including finally jumping back to the address that FPVM has decided upon.

Currently, FPVM eagerly saves and restores the entire GPR and FPR state, FP control state (e.g. `mxcsr`), and `rflags`. It is outside of the scope of this paper, but a possible future optimization might be lazy save/restore of this state. Especially as the floating point state keeps growing (`xsave` can currently occupy a whole page), this might lead to even lower overhead, albeit with a substantial engineering cost in FPVM.

The kernel module exports an `ioctl()` based interface via the `/dev/` filesystem. When FPVM initializes a process, it checks for the existence of this device, and attempts to open it and register its entry point for the user-space entry stub/landing-pad. When FPVM shuts down (or the process crashes), the device is closed, and the process's registration is revoked.

4 Instruction Sequence Emulation

Recall from §2.4 that FPVM decodes, binds, and emulates a single instruction in response to a floating point trap—the instruction that caused the trap to occur. This has a number of advantages, the most important being that only instructions that *can* cause a floating point trap need to be handled. However, this single-instruction-at-a-time model also means that the high cost of trap delivery to the kernel (hw) and from kernel to user (kern) is borne for every single instruction that might trap. Even if we dramatically reduce the hw+kern+ret time (from 5980 cycles to about 760 cycles using trap-short circuiting (§3)), we still have pay that cost for each trapping floating point instruction. The goal of the present optimization is to lower the impact of this cost by emulating multiple instructions per trap, and thus amortizing the cost. Additionally, in deployment environments in which including a kernel module is not allowed, the optimization would be even more valuable because it would amortize the resulting much higher hw+kern costs (5980 cycles).

[3]Red-zone is an x64 ABI optimization that allows the active stack frame to extend below where the stack pointer currently points by 128 bytes. This 128 bytes of potentially live data must be carefully avoided.

4.1 Tradeoffs

While it may seem obvious that this is a good idea, it is important to point out that much depends on the nature of the actual workload, such as the length distribution and popularity of sequences of emulatable instructions, and whether emulating the additional instructions is warranted. To see the latter subtle point, consider a sequence of instructions A,B in which A faults, and we are also able to emulate B. This would let us amortize the fault over two instructions, which is a win. However, suppose that B does not *need* to be emulated. For example, perhaps it involves operands that are not NaN-boxed and the operation does not round. Then we are doing an unwarranted software emulation of B, which will take much longer than simply executing it, which is a loss. In addition, this might generate more NaN-boxed values than are truly necessary.

We also need to consider the software engineering aspects. Suppose we have an instruction sequence X, Y, Z, in which X faults, and Z would also fault, but Y is not a floating point related instruction at all. In order to be able to amortize the fault cost of X over both X and Z, we must implement support for Y. There is a rabbit hole here in which we could easily end up trying to implement a complete system emulator.

4.2 Implementation

We have enhanced FPVM's logic to support multiple instruction sequences. Starting with the faulting instruction, it keeps decoding/binding/emulating instructions until one of the following conditions occurs:

(1) It encounters an instruction for which it has no decoding, binding or emulating support, or
(2) It encounters an instruction it can decode, bind, and emulate, but where no source operand is NaN-boxed.

As the process continues, we update the decode cache with each instruction. This includes placing the sequence-terminating instruction into the decode cache if we have not seen it before, even if case (1) holds. The result is that the decode cache is now effectively a software version of a trace cache [29]. When a sequence/trace is encountered again, each instruction will hit in the decode cache.

Note that (2) implies that it is possible for us to stop emulating on some instruction W, return to the program, and then immediately have W fault because of a Round, Overflow, Underflow, Denorm, or even Invalid event, the latter because the instruction *produced* a NaN. The operation we describe will result in correct behavior, and to truly avoid it for performance reasons would entail us first having to first emulate the instruction W in the IEEE 754 system to see if it would produce any of these effects. We did not think this was worth the engineering effort.

Our implementation also optionally collects detailed statistics on sequences/traces that are actually encountered during execution to generate a detailed profile. This allows us to go far beyond basic amortization benefits in terms of counts and costs, including determining the distribution of sequence lengths, the rank popularity of sequences, the impact of sequences on overall performance (combining sequence length and rank popularity) and the individual sequences themselves, showing the assembly instruction of each sequence, including the terminating instruction and why it terminated the sequence. § 6.3 presents a workload characterization using this instrumentation.

Extending the decoder and emulator. To increase instruction sequence length in the face of (1) requires adding support for more instructions. We used our profiling tool to identify critical instructions to add. Our current design essentially only adds the x64 integer and floating point move instructions. While seemingly simple, there are a vast set issues that arise in these, including different source and destination sizes, sign extension versus zero-extension, integer/floating point conversion due to moves between GPRs and FPRs, string moves, as well as untangling move functionality that is slightly different in different extension families. We implemented support for about 40 move opcodes, and decided to ignore (for now) an additional 123. We also currently ignore all control flow instructions, so our traces are bounded by basic block size, but this does not seem to be a major blocker.

Compared to the baseline, we also added support for the `cmpxx` family instructions (e.g., `vcmpltsd`), but this was largely because the initial FPVM codebase we started from omitted these because testing was done on a target platform that did not produce them.

Arguably, this exercise would be considerably simplified on a RISC architecture, such as RISC-V or ARM.

5 Kernel-bypass for correctness instrumentation

Recall from §2.6 that correctness instrumentation is introduced into the binary being virtualized via a static analysis process that patches the binary to handle flows of potentially NaN-boxed floating point values to non-floating point contexts (*memory-escape correctness*), and to handle flows of arguments into functions in shared libraries that are not subject to the analysis (*foreign function correctness*). The latter occurs due to the engineering challenge of having all binary code available (or having a large application be statically linked). The static analysis and both correctness traps could be avoided if the x64 floating point hardware were fully virtualizable.

In the baseline system, both forms of correctness instrumentation are handled by having the analysis place `int3` instructions before the problematic instructions. These invoke FPVM via hardware dispatch of a breakpoint trap and kernel signal delivery of `SIGTRAP`, and allow it to demote values if necessary. The goal of the present optimization is to avoid both the hardware and kernel components. Because FPVM is part of the process and linked into its ELF chain, we can, in principle, replace the `int3` with a direct call to the correctness handler. This would reduce overheads by several thousands of cycles for each correctness trap, as the program would not have to transition through the kernel. Unfortunately, this is not as trivial as inserted calls are later in the ELF chain than FPVM, and thus cannot see symbols exported by FPVM.

5.1 Profiling Instead of Static Analysis

The previously reported method of finding instructions which require *memory escape traps* for correctness required a binary static analysis. This analysis worked at an instruction level, and performed Value Set Analysis to find which integer instructions a floating point value *might* flow into through memory. This analysis is equivalent to alias analysis, and its runtime and memory demands tend to explode. For example, Enzo takes multiple days and requires terabytes of swap to statically analyze. This makes it difficult to use the prior version of FPVM on large applications.

To address this problem, we have written a new analysis which utilizes profiling instead of binary analysis. We built a tool on top of PIN [25] which instruments all memory operations in the application. When a floating point value is stored to memory[4], the eight byte block of memory it is stored to is marked as "containing a float". If an integer instruction then reads from any location marked as containing a float, that instruction is added to the set of instructions that need to be patched. The profiler also unmarks memory blocks as containing floats in several situations: stack unwinding, storing an integer, etc. The output from the profiler and analysis are roughly the same—a set of integer instructions which must have any potential FPVM-owned NaN-boxed values in the destination registers demoted before they can they can continue. The profiler will identify fewer instructions, however, because it is dynamically considering the flows in a specific run instead of statically considering all possible flows.

Our profiler allows developers to patch their application for FPVM by simply profiling it with the same workload, which takes *much less* time than the previous static analysis. We use this profiler throughout the evaluation of the paper, as it does not produce different performance results compared to the prior binary static analysis.

Note that, due to the memory and time requirements of the previous static-analysis approach, we were unable to use it as a baseline and instead only use our profiling approach in our evaluation. A comparison of the results from original FPVM paper [15, Figure 9] with Figure 1 readily reveals that "correctness overhead" has been nearly entirely eliminated across all the cases where it was previously significant. This is because the rate of correctness traps is much lower.

5.2 Magic traps for Memory-Escape Correctness

Before executing an instruction which uses a floating point value in a non-floating point way (for example: interpreting a floating point value as an integer to extract the sign bit) that value must first be demoted from the NaN-boxed representation back to a float. This kind of reinterpretation is usually done through an *escape to memory*, where a float is stored to memory, and an integer is loaded. As illustrated in the left half of Figure 3, the previous approach used e9patch to introduce an `int3` instruction before the troublesome

[4]x64 is surprisingly well typed, and compilers will emit instructions which are tagged as "scalar double" for double stores (*movsd*)

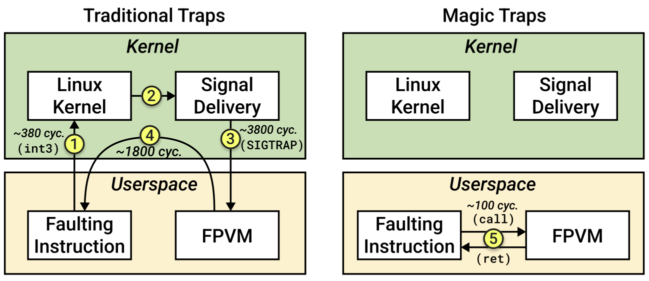

Figure 3: Magic traps bypass the kernel, saving thousands of cycles per trap, when memory correctness traps are needed

instruction to trigger a trap to kernel mode ①. The kernel then dispatches this trap ② through a slow general purpose interface and delivers a `SIGTRAP` to FPVM ③ which calls the function to demote the troublesome value before returning through a `sigreturn` ④.

Our approach to accelerating this instrumentation, "magic traps", is outlined in the right half of the figure. The main difference is that instead of an `int3` instruction, the patcher instructs e9patch to insert a `call` before the offending instruction which invokes a "trampoline" function which is linked into the patched binary. Because this call is patched into the binary at unknown locations—where a call was not originally planned—this trampoline function must take care to not clobber any values in registers or on the stack. It first shifts the stack pointer past the red-zone and saves all registers similar to a context-switch. Because of the placement on the ELF linker chain the trampoline cannot directly invoke FPVM and upon first invocation must rendezvous with the FPVM runtime to call the demotion handler. To that end, when FPVM starts it maps a "magic page" into the process address space at a well known adress (similar to VDSO) and populates it with a cookie and the address of the demotion handler function. The trampoline function then reads this magic page and recovers the pointer to the callback function. Subsequently, all patched instructions invoke FPVM through this pointer. In this way, the hardware trap (380 cycles), signal dispatch (5600 cycles with `SIGTRAP` + `sigreturn`, 350 cycles with the kernel module) costs are replaced with the cost of what is effectively a double-indirect call and return (about 50 cycles) ⑤. These "magic traps" reduce the cost of getting into FPVM to deal with memory escape correctness by a 14-120x. In combination with the improvements brought by the profiler, magic traps effectively eliminate the overhead of memory escape correctness instrumentation.

5.3 Magic wraps for Foreign Function Correctness

Because the static analysis pass which identifies memory escapes cannot run on functions from external libraries such as libc, we must conservatively assume that those functions *will* interpret floats as integers. As a motivating example, consider the library function `printf` which, when printing a floating point value, will interpret the *bits themselves* to determine features such as the sign of the floating point number. On x86 these operations cannot cause a trap, and are thus lost to FPVM, leading to incorrect program output.[5] To handle this we use `LD_PRELOAD` wrappers, where a wrapper stub function for some shared library function is injected into the link chain (`printf()` for example). Because of its appearance earlier in the dynamic link order, the application will invoke the FPVM stub instead of the "real" function.

We have implemented two approaches to function wrapping. The first is to generate these wrapper functions as assembly code with callbacks to FPVM functions written in C that perform preprocessing (demote argument register contents, set `mxcsr`, etc)[6]. These wrapper functions are constructed using carefully crafted assembly which manages the stack frame so that from the perspective of the wrapped function (e.g. the real `printf`) the wrapper function's stack frame does not exist. This is necessary to correctly handle cases where arguments are passed via the stack. Additionally, the wrapper must carefully manage its own use of registers as to not clobber any arguments or callee saved register values so they are not changed on entry to the wrapped function. The problem with this approach is that any function which is wrapped cannot be used from within FPVM itself. For example a call to a wrapped `printf` from within FPVM will invoke another part of FPVM which could itself call `printf`.

The second approach, which we call "magic wrapping", generates these wrapper functions in the same way as the native approach, but places them in a different namespace entirely (e.g. `printf` will have a wrapper named `printf$fpvm` generated). We then use Lief [2] to modify the *program's* symbol table to point to these new wrapped functions instead. This namespacing ensures that wrapped functions are totally independent of FPVM's own namespace, which means FPVM does not need to carefully control its own use of symbols.

The `libm` functions are always configured with special handwritten forward wrappers that interface with the alternative arithmetic system. We use magic wrapping in our evaluation. Because the wrapper functions and how/when they are called are identical, there is no performance difference compared to forward wrapping.

6 Evaluation

We now evaluate the effectiveness of the aforementioned techniques to accelerate floating-point virtualization by reducing the cost of trap delivery with trap short circuiting and sequence emulation. We test FPVM against a mix of benchmarks and applications. Our test code consists of the FBench floating point benchmark suite [33], the FFBench fast Fourier transform benchmark [34], a version of the Lorenz system simulator that we developed, and a three-body problem simulation. We also evaluate FPVM against Enzo [9], an astrophysics and hydrodynamics simulator written in about 307,000 lines of C, Fortran, and Python. Because we are using the "Boxed IEEE" alternative arithmetic implementation for all of these tests, we expect to get bit-for-bit equal results to the baseline, and we have validated this to be true.

All testing was conducted on a Dell R6515 with an AMD EPYC 7443P 24-Core Processor capable of boosting to 4GHz with 512GB of DDR4 3200MT/s memory in one NUMA domain. This processor supports all floating point extensions besides AVX-512. The machine runs Ubuntu 22.04 (jammy) with Linux kernel version 5.15.

[5]Often, this results in the program printing "nan" or "-nan" to the terminal

[6]The postprocessing step does not need to promote anything as all floating point registers are defined by the ABI as being caller-save.

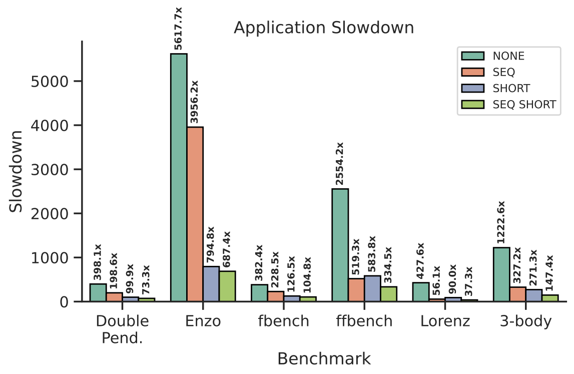

Figure 4: Using the techniques described in this paper, the end-to-end runtime slowdown of floating point virtualization, as exemplified by FPVM, has been reduced by an order of magnitude. Note that this overhead includes the intrinsic slowdown of the worst-case alternative arithmetic library.

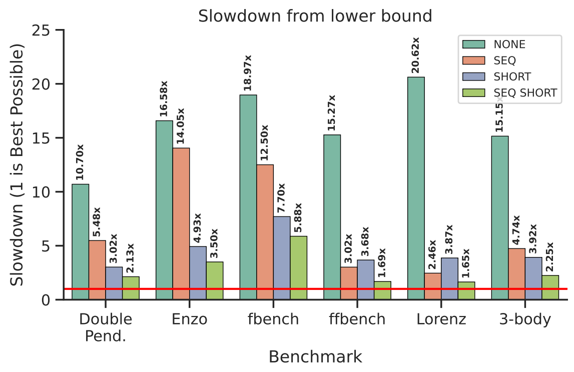

Figure 5: Slowdown of FPVM relative to the intrinsic slowdown of the worst-case alternative arithmetic library. 1x slowdown (red line) is the best possible, meaning there is no virtualization overhead on top of altmath. Our optimizations allow FPVM to approach this limit.

GCC 11.4 was used to compile all code. With this evaluation, we seek to answer the following research questions:

- **Q1**: How do the acceleration techniques outlined affect the overheads of FPVM? How close are these overheads to the lower bound? (§6.1)
- **Q2**: To what degree do the acceleration techniques reduce virtualization overheads for any given instruction? (§6.2)
- **Q3**: How effectively does sequence emulation amortize the cost of floating-point traps, and what are the practical constraints on trace cache size needed to achieve significant performance gains? (§6.3)
- **Q4**: How do these techniques extend to a more complex alternative arithmetic system (e.g. MPFR)?

6.1 Our Acceleration Techniques Reduce the Overhead of Virtualization Considerably

To answer **Q1**, we ran the applications mentioned above both with and without FPVM, for each of the combinations of our acceleration techniques, measuring wall-clock execution times. Figure 4 shows the results of this test.

The baseline configuration with no acceleration techniques (*NONE*) shows considerably high slowdowns up to 5617x. This is in-line with the scale of the findings from the original FPVM paper[15], albeit on a much faster machine with vastly different performance characteristics. The second bar, *SEQ*, shows the absolute slowdown of FPVM with *only* sequence emulation applied, which provides a moderate reduction in slowdown. Trap short circuiting (*SHORT*) shows a much larger overhead reduction, and together they combine to only a net positive benefit. The main takeaway from this evaluation is that the acceleration techniques can achieve an average of 7.2x reduction (11.5x in the case of Lorenz) in slowdown compared to a version of FPVM without them.

One might comment that the overheads of FPVM are still considerable: 37.3x in the best case is still a considerable overhead. An obvious question to ask, then, is what is the lower-bound overhead for each application and how close is our accelerated FPVM to achieving is? We consider the lower-bound overhead of floating point virtualization to simply be the overhead of running the floating point operations through the alternative arithmetic library (i.e. the intrinsic costs outlined in §2.7). This lower-bound is approximating the slowdown of an FPVM implementation which has *zero* virtualization overhead.

Recall that in this paper we are focusing on a "worst-case" alternative arithmetic library, "Boxed IEEE", which simply NaN-boxes double precision values in the heap. We consider this to be worst-case because it is the fastest possible NaN-boxed representation.

To evaluate this, we configured FPVM to record fine-grained performance and telemetry information to determine the total time spent executing the alternative arithmetic operations over the course of application execution. We then add that time to the baseline without running FPVM from Figure 4 and use that as a new baseline in Figure 5. As can be seen, with no acceleration (*NONE*), FPVM has considerable slowdown on top of the alternative math. This clearly tracks from Figure 1, where the **altmath** component is a tiny contributor to the overall costs. The majority of this non-intrinsic overhead comes from virtualization overheads: trapping, dispatching, decoding, and emulation.

Once acceleration techniques are added to reduce these non-intrinsic costs (*SEQ* and *SHORT*), the cost of FPVM shrinks considerably. Lorenz, for example, is just 1.65x slower than if the application simply used NaN-boxed floating point values with no virtualization overheads. Some applications have less benefit from these acceleration techniques—notably Enzo does not benefit as greatly from sequence emulation as other workloads do—but overall the takeaway is that the acceleration techniques outlined have brought us *an order of magnitude closer to eliminating virtualization overheads.*

6.2 Amortization Analysis

To answer **Q2**, we configured FPVM the same way as it was to gather data for Figure 1, plus additional low cost performance profiling code, and ran it on our benchmarks. The results can be seen in Figure 6. As before, *NONE* has no acceleration, *SEQ* includes sequence emulation, *SHORT* includes trap short-circuiting, and *SEQ*

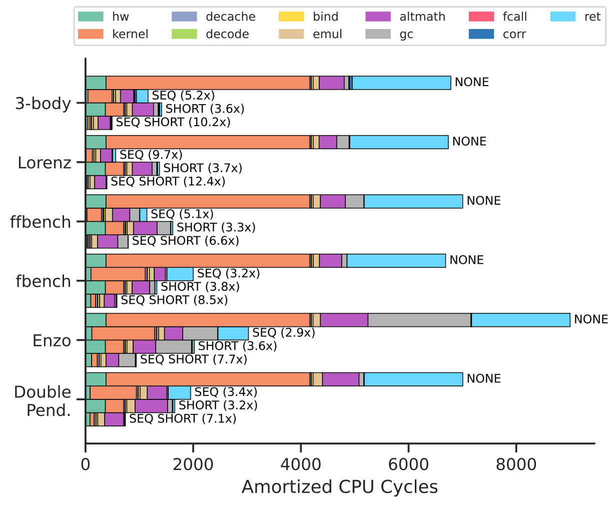

Figure 6: Breakdown of costs with trap short circuiting and sequence emulation enabled (amortized to the cost per emulated instruction) in FPVM using Boxed IEEE as the alternative arithmetic system. Compare with Figure 1. As the optimizations are applied, altmath becomes an ever larger component of the overhead, approaching the Amdahl limit.

SHORT includes both. The goal is to make the **altmath** component a much larger proportion of the overall overhead, and, ideally, to dominate it.

Our magic trap and wrap acceleration techniques are always enabled in these breakdowns. Our new profiling technique for identifying problem spots (§5.1) results in far fewer required correctness events to process in our benchmarks and as a result the effect of magic trap and wrap is much less than would be expected given the original FPVM paper's results. Consequently, we decided to remove magic trap and wrap from this breakdown. It is however important to understand that an application that requires many correctness events would see significant benefit from magic trap and wrap.

Answering **Q2**, trap short-circuiting (*SHORT*) considerably reduces the trap overhead (kern, ret) in all cases. Sequence emulation (*SEQ*) amortizes trap overheads (hw, kern, ret) by an amount proportional to the average number of instructions in a sequence. Lorenz, for example, handles ~32 instructions per trap on average, reducing the amortized hw+kern+ret overheads by 32x. Enzo benefits less from this, as its average sequence length is only ~3.

These two techniques are synergistic and by combining the two, the best of both worlds can be achieved. The shorter the average sequence length, the less we benefit from sequence emulation and the more we benefit from trap short-circuiting. At the happy extreme, applications with long sequences reduce the already-low trap times from *SHORT* to nearly nothing (in Lorenz, down to ~10 cycles per instruction), and returning from FPVM is practically free.

Most importantly, combining trap short circuiting, sequence emulation, our new profiling technique, and the trap and wrap optimizations results in each benchmark/application now spending a vastly larger portion of its time in *altmath*, making it the (unavoidable) bottleneck in most cases. This lines up with the results

```
  addsd   xmm12, xmm5
  movsd   xmm5, qword ptr [rip + 0x91d]
  ...
  movapd  xmm0, xmm8
* movhpd xmm11, qword ptr [rsp + 0x30]
* mulsd  xmm4, xmm15
```

Figure 7: Example instruction trace.

from §6.1—the higher proportion of overheads that *altmath* is, the closer the app is to the lower bound in Figure 5.

6.3 Sequence Emulation Analysis

As we described in §4, our sequence emulation system can be configured to capture detailed statistics on sequences we encounter during execution, and then dump the resulting profile at the end of the run. Long, popular sequences result in better amortization of the trap cost by increasing the average number of emulated instructions per floating point trap. Does this happen? How big do we need to make the trace cache? Is it too big to be practical? What is the performance benefit of the amortization?

Figure 7 gives an example instruction trace, the third most popular trace in the Lorenz Attractor benchmark. It accounts for about 11% of all encountered traces in the benchmark (of any length). It is 15 instructions long, though the figure elides some of these for brevity. The very first instruction (`addsd xmm12, xmm5`) is the one that caused the floating point trap that invoked FPVM. The first asterisked instruction (`movhpd xmm11, qword ptr [rsp + 0x30]`) is the instruction that terminated the sequence. Here the `movhpd` ("move high packed double") is loading a double from memory and replacing the high-order double in the `xmm11` register with it, leaving the low-order double unmodified. We felt that adding support for partial vector moves (on top of normal scalar and vector moves) was unnecessary engineering effort.[7] The hardware will execute the `movhpd` without any floating point trap. The very next instruction (`mulsd xmm4, xmm15`) is one that we *do* support, Thus, if we were to add support for the family of `movhpd` instructions, we would not just extend this sequence length by one, but by at least two (the next instruction is not captured.)

Given all the captured traces and their statistics, and being mindful that the trace lengths depend on our current implementation, we can ask about the necessary trace cache size. Our decode cache, which we use to store traces, defaults to 64K instruction entries, with ≤ 1024 bytes per entry, and thus cannot exceed 64 MB. For the runs in this paper, it is always much smaller than this, with less than 2,000 entries for the largest case (Enzo). This should not be particularly surprising, as one would expect the action to be concentrated in hot loops.

Of course, it could be that we need to cover many sequences in order to put a dent into the emulated instructions, which could be a problem. To assess this, Figure 8 shows the rank popularity distribution of the traces we have captured, with one curve per benchmark/application. The more the CDF is skewed to the left, the better a cache can perform. Clearly, there is a difference between the benchmarks (left cluster of curves) and the Enzo application

[7]This family of load instructions also has different semantics depending on whether it is legacy SSE2 (as here) or VEX+EVEX encoded (multiple overwrites)!

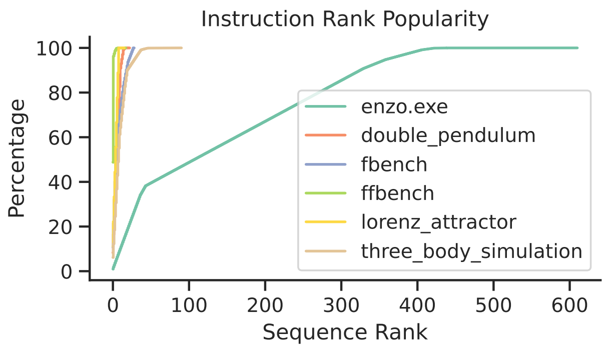

Figure 8: Instruction sequence rank popularity in terms of number of emulated instructions.

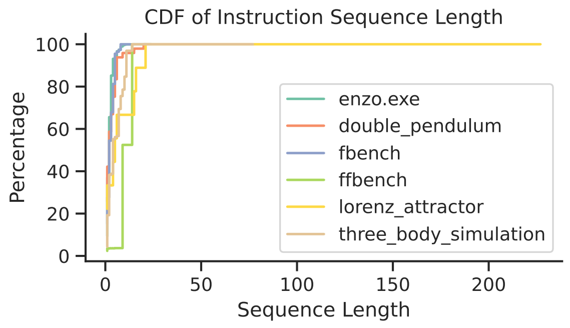

Figure 9: Instruction sequence length distribution.

(right curve). Fewer than 100 sequences are needed to cover the benchmarks. Consider Enzo at rank 350. The figure is saying that caching the 350 most popular sequences in Enzo will result in covering about 90% of its emulated instructions. There are a little over 600 total sequences in Enzo, which is still a tiny number.

Ideally, we would like instruction sequences to be as long as possible, since that maximizes amortization. At the same time, the longer the sequences are, the more space the trace cache will require. Figure 9 shows the length distribution of traces that we captured, with one curve for each benchmark/application. We would prefer these curves to skew to the right. As can be seen, the distributions vary widely, though this is slightly hidden by Lorenz having an extremely long (albeit unpopular) instruction sequence. It is also important to note that the distribution will depend a great deal on compiler optimizations used to compile the program. For example, loop unrolling and software pipelining optimizations will naturally lead to longer sequences.

Figure 10 attempts to show the combined effect of sequence rank popularity and sequence length. The question the figure addresses is "if we only were able to cache the top-k most popular sequences, what would the average sequence length be?" So, for example, if we consider the top-20 sequences in fbench, the average sequence length is a little over four, while if we consider just the top-5 sequences in Lorenz, the average sequence length is already 30. Enzo's long tail here is simply due to having many more sequences. With the top-300 sequences, the average sequence length is about 3.

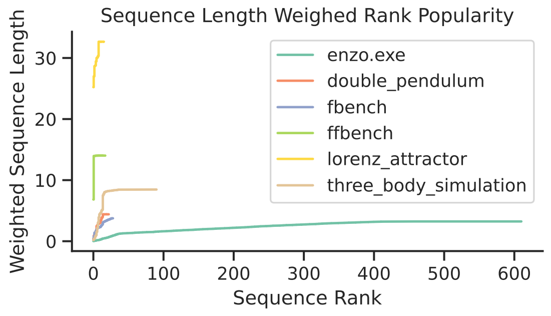

Figure 10: Instruction sequence length weighted rank popularity. Each trace converges on the average sequence length encountered in our performance evaluation.

Each trace in the figure converges to the average sequence length we encountered in our tests, where we did not limit the trace cache. Note that this tells a story of its own. The lower the average sequence length is, the more critical *other* optimizations are. In Lorenz, we execute an average of 32 instructions per floating point trap, while in fbench, we barely manage 4. The rank at which we have convergence, multiplied by the average instruction length at that rank also tells us the cache size. So, for example, Lorenz needs $18 \times 32 = 576$ entries, or about 576 KB of trace cache space. The worst case, Enzo, is $600 \times 3 = 1800$ entries or about 1.8 MB.

Answering **Q3**, the longer the average sequence length, the more we amortize the costs of hw, kern, and ret. Without sequence emulation, this length will be one across the board.

6.4 Performance with MPFR

Thus far, we have evaluated the accelerated version of FPVM on a "worst case" alternative arithmetic system – Boxed IEEE. This implementation has the lowest overhead of any system offered by FPVM, exacerbating the virtualization overhead of FPVM itself. Figure 11 and 12 show the same workloads as Figure 4 and 5, except they are run using MPFR instead of Boxed IEEE.

Switching to MPFR is straightforward–FPVM is simply reconfigured in seconds to use it with 200 bits of precision. Figure 11 shows that overheads are higher than with Boxed IEEE because math in MPFR is more expensive to perform. On the other hand, the slowdown under FPVM is much closer to the intrinsic slowdown of using MPFR, as shown in Figure 12. As the intrinsic slowdown of the alternative arithmetic system grows, the slowdown under FPVM approaches it. Figure 13 shows the amortized cost per instruction under MPFR. As would be expected, a larger portion of the cost is due to **altmath** as MPFR itself is more expensive than Boxed IEEE. Note that MPFR has slightly higher "gc" overhead particularly in Enzo. This is due to MPFR allocating more temporary objects than Boxed. This presents an easy point of optimization in future work.

7 Related Work

Alternative floating point arithmetic systems have gained increasing attention as researchers seek to improve numerical stability,

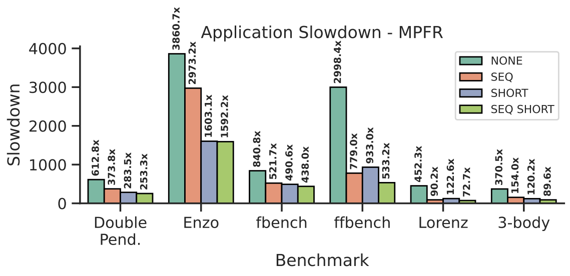

Figure 11: Switching to MPFR is straightforward, and slowdown under MPFR benefits from our techniques.

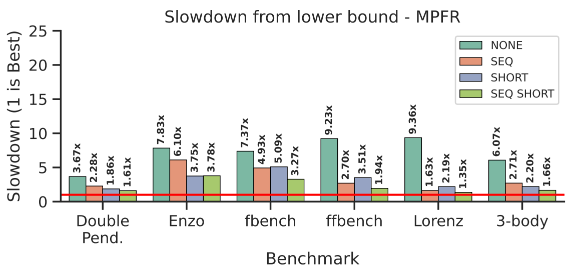

Figure 12: FPVM slowdown with MPFR approaches the intrinsic slowdown of MPFR itself.

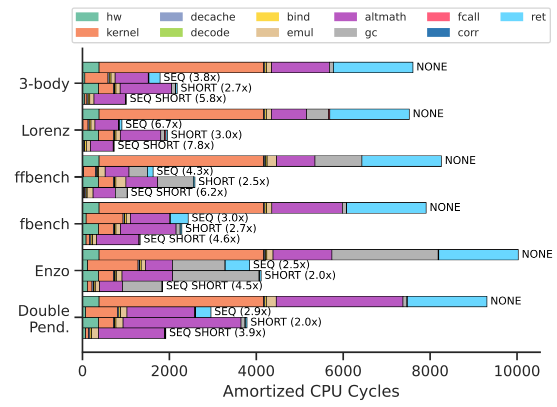

Figure 13: The amortized cost per instruction is generally dominated by MPFR (altmath).

precision, and performance in various applications. These efforts can be categorized into two main directions: the development of alternative representations which can provide different precisions, and tools for analyzing and improving floating point accuracy.

Several alternative number representations have been proposed to address the limitations of IEEE 754 floating point arithmetic. These include unums and posits [19, 21], BFloats [22], GNU MPFR [18], libBF [5], and logarithmic arithmetic [3], all of which offer improved precision, dynamic range, or efficiency in specific workloads such as machine learning. Other approaches, including slash arithmetic [26] and interval arithmetic[20], aim to provide error bounds and improved numerical stability. However, using such libraries requires modifying source code, which can be challenging for large or legacy applications. Some researchers advocate for a complete rethinking of floating point arithmetic in favor of an API to the real numbers[8], which would allow programmers to reason about computations using standard mathematical rules while achieving practical performance. This approach (or higher precision) might also mitigate the effects of misunderstandings developers have about various aspects of floating point [13, 14].

A range of tools aim to improve source code quality by identifying sections with high dependence on precision or compiler/hardware optimizations, which may cause numerical stability issues due to algorithmic design or buggy optimizations that alter semantics [4, 6, 7, 11, 12, 17, 23, 24, 27, 28, 30–32]. Many of these tools use shadow arithmetic with different precision than the original code, and some operate directly on application binaries, avoiding the challenges of source-level approaches. While they often reduce performance, their ability to quickly build code coverage mitigates this concern. Our work on floating point virtualization addresses these challenges by allowing existing, unmodified binaries to use alternative arithmetic systems without recompilation. Unlike existing approaches that require source code modifications or recompilation, our approach operates at runtime, making it more practical for large-scale scientific applications. Additionally, we address the performance bottlenecks inherent in traditional floating point virtualization, significantly reducing its overhead and making it a viable alternative for real-world workloads.

8 Conclusions and Future Work

Floating point virtualization has the ability to enable the use of alternative arithmetic systems without requiring source code changes. However, its practicality has been limited by high overheads in the traditional trap-and-emulate model, where trap delivery mechanisms introduce significant virtualization overheads. In this work, we introduced trap short-circuiting, instruction sequence emulation, and kernel-bypass for correctness instrumentation—three techniques that dramatically reduce the cost of virtualization. Our implementation within FPVM on x64/Linux achieves a 10x reduction in per-instruction overhead, bringing virtualization performance within 1.65x of the lower bound set by the worst case alternative arithmetic system for FPVM. With a more realistic MPFR system, FPVM can achieve 1.35x of the lower bound.

We are currently working towards architecture-independence, with immediate additional support for ARM. Conceivably, FPVM could operate on any environment that can provide floating point traps, including some GPUs such as AMD. We are also extending the RISC-V architecture in several ways to better support floating point virtualization, including adding very fast floating point trap support, and hardware support to replace correctness traps.

Acknowledgments

We thank the anonymous reviewers for their time and feedback. We also thank members of the Prescience Lab for their support and feedback on this work. This effort is based upon work supported by the U.S. National Science Foundation (NSF) under awards CNS-2211315, CCF-2119069, and CNS-2211508.

References

[1] 2021. *Capstone: The Ultimate Disassembler.* Retrieved Jan 19, 2019 from https://www.capstone-engine.org/

[2] 2025. LIEF - Library to Instrument Executable Formats. https://github.com/lief-project/LIEF/.

[3] M. G. Arnold, T. A. Bailey, J. R. Cowles, and J. J. Cupal. 1990. Redundant Logarithmic Arithmetic. *IEEE Trans. Comput.* 39, 8 (Aug. 1990), 1077–1086.

[4] 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).*

[5] Fabrice Bellard. 2017. LibBF: The Tiny Big Float Library. Available at https://bellard.org/libbf/.

[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] Hans-J. Boehm. 2020. Towards an API for the Real Numbers. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[9] Greg L. Bryan, Michael L. Norman, Brian W. O'Shea, Tom Abel, John H. Wise, Matthew J. Turk, Daniel R. Reynolds, David C. Collins, Peng Wang, Samuel W. Skillman, Britton Smith, Robert P. Harkness, James Bordner, Ji hoon Kim, Michael Kuhlen, Hao Xu, Nathan Goldbaum, Cameron Hummels, Alexei G. Kritsuk, Elizabeth Tasker, Stephen Skory, Christine M. Simpson, Oliver Hahn, Jeffrey S. Oishi, Geoffrey C. So, Fen Zhao, Renyue Cen, Yuan Li, and The Enzo Collaboration. 2014. ENZO: An Adaptive Mesh Refinement Code for Astrophysics. *The Astrophysical Journal Supplement Series* 211, 2 (2014), 19.

[10] Annie Cherkaev. 2018. The Secret Life of a NaN. https://anniecherkaev.com/the-secret-life-of-nan.

[11] 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.

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

[13] 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.

[14] 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).*

[15] Peter Dinda, Nick Wanninger, Jiacheng Ma, Alex Bernat, Charles Bernat, Souradip Ghosh, Christopher Kraemer, and Yehya Elmasry. 2022. FPVM: Towards a Floating Point Virtual Machine. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing* (Minneapolis, MN, USA) *(HPDC '22).* Association for Computing Machinery, New York, NY, USA, 16–29. https://doi.org/10.1145/3502181.3531469

[16] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary Rewriting without Control Flow Recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020).* Association for Computing Machinery, New York, NY, USA, 151–163. https://doi.org/10.1145/3385412.3385972

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

[18] 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).

[19] John Gustafson. 2015. *The End of Error: Unum Computing.* Chapman and Hall/CRC.

[20] T. Hickey, Q. Ju, and M. H. Van Emden. 2001. Interval Arithmetic: From Principles to Implementation. *J. ACM* 48, 5 (Sept. 2001), 1038–1068.

[21] Willian Kahan. 2016. A Critique of John L. Gustafson's The End of Error—Unum Computation and his A Radical Approach to Computation with Real Numbers. In *Proceedings of the 23rd IEEE Symposium on Computer Arithmetic (ARITH).*

[22] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep DubeyAbhisek Kundu. 2019. A Study of BFLOAT16 for Deep Learning Training. arXiv preprint arXiv:1905.12322.

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

[24] 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).*

[25] 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. *SIGPLAN Not.* 40, 6 (June 2005), 190–200. https://doi.org/10.1145/1064978.1065034

[26] D. W. Matula and P. Kornerup. 1985. Finite Precision Rational Arithmetic: Slash Number Systems. *IEEE Trans. Comput.* C-34, 1 (Jan 1985), 3–18.

[27] 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.

[28] 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).*

[29] Eric Rotenberg, Steve Bennett, and James E. Smith. 1996. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 1996).* 24–35.

[30] 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).*

[31] 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).*

[32] 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.

[33] John Walker. 2021. FBench: Floating Point Benchmarks. https://www.fourmilab.ch/fbench/.

[34] John Walker. 2025. FFBench: Fast Fourier Transform Benchmark. https://www.fourmilab.ch/fbench/ffbench.html.

[35] A. Wingo. 2011. Value Representation in JavaScript Implementations. http://wingolog.org/archives/2011/05/18/value-representation-in-javascript-implementations.