Enabling Floating Point Virtualization With Tiny Numbers

Kevin Hayes
Northwestern University
Evanston, Illinois, USA
KevinHayes2026@u.northwestern.edu

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

Abstract

Floating point virtualization allows existing, unmodified application binaries to be run using an alternative arithmetic system. Such virtualization is geared to alternative numbers that are "larger" (require more bits) than the IEEE 754 numbers (e.g., 64 bit doubles) they replace. In this work, we approach the challenge of virtualizing with "smaller" numbers (requiring fewer bits), which is of increasing interest given the explosion of low-precision hardware targeting AI. We focus specifically on the ubiquitous x64 architecture through a hardware/software co-design that leverages x64 functionality that currently lays fallow. The design combines (a) instruction traps via lazy FPU abduction, and (b) simplified memory management by tiny value boxing. We also develop an example tiny alternative arithmetic system that allows smaller IEEE 754 numbers, down to 3 bits, with the exact precision able to be specified on a per-value or per-instruction basis at runtime. Our prototype system is evaluated using validation and performance tests based on running NAS and other benchmarks with a range of lower precision numbers.

CCS Concepts

• **Software and its engineering** → **Operating systems**; **Virtual machines**; • **Mathematics of computing** → *Mathematical software performance*; *Numerical analysis*; **Arbitrary-precision arithmetic**.

Keywords

virtualization, floating point arithmetic, software development, IEEE 754

ACM Reference Format:
Kevin Hayes and Peter Dinda. 2026. Enabling Floating Point Virtualization With Tiny Numbers. In *The 35th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '26), July 13–16, 2026, Cleveland, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3806645.3807578

1 Introduction

The goal of floating point virtualization is to enable unmodified application binaries to be run using alternative arithmetic systems. The alternative arithmetic system is introduced at the *level of machine instructions*, making such virtualization oblivious to language and thus very easy to introduce. In effect, floating point virtualization augments the architecture, showing the developer the effect of an architecture that included the alternative arithmetic system instead of IEEE 754 floating point arithmetic [20–22].

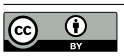
Floating point virtualization can be implemented on x86 processors using a trap-and-emulate virtualization strategy (similar to a traditional OS-level virtual machine monitor) [13], and can be optimized to achieve overheads that approach those of using the alternative arithmetic system itself [38]. Other architectures, such as ARM and RISC-V, can also support floating point virtualization [16]. The core required functionality is that the hardware to be able to trap when one of exceptional conditions identified in the IEEE 754 standard occurs.

Motivation. Alongside the considerable long-standing interest in *higher precision* alternative arithmetic systems, such as GNU MPFR (arbitrary precision IEEE 754) [14], libBF [4], rationals [30], intervals [17], Posits/Unums[15, 26], interfaces to the reals [5], etc, there is also an ongoing explosion of *lower precision* arithmetic systems, such as BFloats [27], FP16 (part of IEEE 754, 2008), FP8 [31], INT8 [40], NF4 [9], logarithmic arithmetic [2], etc, that is being driven by the giant market that is AI/ML.

There is considerable interest in using the hardware created by the AI/ML segment for high-performance scientific computing as well. This leads us to a conundrum: AI prefers greater amounts of smaller number representations while scientific workloads demand the rigors of high-precision numbers. How well can lower precision, machine learning workload-driven alternative arithmetic systems work for this domain? Floating point virtualization could enable a developer to perform initial testing on existing applications with minimal effort.[1] This interest in considering the implications of lower precision for scientific computing has produced a range of work, including Precimonious [35], VPFloat [24, 25], shadow values in NSan [8], and most recently RAPTOR [18, 19] that all can lower precision or otherwise use alternative arithmetic systems through the use of compile-time transformations in LLVM. Our work differs in that floating point virtualization targets the **unmodified binary** and is essentially an operating systems-level approach to the problem.

Limitations of state-of-the-art approaches. Unfortunately, existing floating point virtualization based on trap-and-emulate cannot currently support such lower precision alternative arithmetic systems. Floating point virtualization relies on using IEEE 754 hardware execution as its baseline. The virtualization software is only invoked when the hardware signals that an exact representation of an instruction's result is not possible (e.g., by signaling a rounding exception that triggers a floating point trap). When using lower

[1] We hasten to add that in many cases numerical methods will also need to be adapted to the alternative arithmetic system. Floating point virtualization would provide the ability to do a "dry run" without any significant effort. The results of this would then feed into the decision-making process to determine whether the larger effort is warranted.



This work is licensed under a Creative Commons Attribution 4.0 International License.
HPDC '26, Cleveland, OH, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2640-8/26/07
https://doi.org/10.1145/3806645.3807578

precision numbers, we need to trap well before this point because a value that can be exactly represented in IEEE 754 may not be exactly representable in the lower precision system.

Certainly a lower precision alternative arithmetic system can be plugged into trap-and-emulate-based floating point virtualization, but what portion of the application's dynamic instructions actually run under the lower precision is unknown and difficult to determine—*the hardware cannot tell us which dynamic instructions did not fault.*

Key insights and contributions. We eliminate this limitation of floating point virtualization, enabling its correct use with lower precision alternative arithmetic systems. More specifically, we show how to ensure that *all* dynamic floating point instructions use the lower precision alternative arithmetic system on the x64 architecture. Our techniques are potentially generalizable to other architectures provided they have a specific feature.

We leverage x64 hardware support, specifically the `cr0.ts` bit and the corresponding `#NM` hardware fault. The original motivation for this hardware support was to enable "lazy FPU context switching". Because the floating point state is large and growing, the idea was to avoid saving and restoring it on context switches. When switching back and forth between a thread using floating point and others that are not, the hardware support makes it possible for the kernel to completely avoid saving and restoring FPU state, while maintaining correctness.

Modern kernels emphatically do *not* use this feature, for two reasons. First, the "FPR state" (e.g., `xmm`/`ymm`/`zmm` registers, `mxcsr`, etc) has been extensively leveraged to enable *non*-floating point instructions. These in turn are widely leveraged by low-level library programmers and modern compiler code generators to accelerate work that would historically only have used instructions that touch "GPR state". Even a simple `memcpy()` today is almost certainly using vector instructions that use the "FPR state". Consequently, there are very few threads that do not touch the "FPR state" between context switches, obviating the benefits of lazy FPU context switching. The second reason is that the lazy FPU context switching technique has been shown to have side-channel vulnerabilities, so carrying over ostensibly "unreadable" "FPR state" from one thread to another can leak information from one process to the other.

We show how to leverage the currently unused `cr0.ts`/`#NM` functionality in a specialized kernel module that solves the problem of the missing floating point traps. Our technique, *lazy FPU abduction* results in all dynamic instructions in the program that use "FPR state" trapping to the virtualization software. The virtualization software then emulates the floating point instructions and allows the non-floating point instructions to execute as normal.[2] Consequently, the need to emulate all floating point instructions to correctly virtualize using lower precision alternative arithmetic systems is met. Lazy FPU abduction is likely applicable to other architectures that support or have supported lazy FPU context-switching.

Our broader contributions are as follows:

- We describe the issues that arise when extending floating point virtualization to lower precision alternative arithmetic systems.
- We address a key issue via the lazy FPU abduction technique described above, providing a kernel module for x64 that can be used with an off-the-shelf Linux kernel.
- We show how lower precision numbers eliminate the garbage collection problem in floating point virtualization because the numbers can be "value boxed" into NaNs.
- We develop a tiny alternative arithmetic system that can support variants of IEEE 754 that operate on a wide range of precisions, from 3 bit values (sign bit, exponent bit, mantissa bit) to 50 bit values (sign bit, 11 exponent bits, 38 mantissa bits).
- We use the above contributions to add low precision alternative arithmetic to the publicly available FPVM floating point virtualization system.
- We provide mechanisms within our low precision alternative arithmetic system to test mixed-precision approaches using virtualization.
- We analyze the effect on program behavior and measure the overheads of this approach by virtualizing various benchmarks at low precision.

Experimental methodology and artifact availability. The bulk of the paper describes our methodology, which is centered around a proof-of-concept implementation of lazy FPU abduction, an extension of FPVM to support lower precisions and integrate with the lazy FPU abduction kernel module, and evaluation with a range of workloads. Our kernel module and extensions to FPVM will be made available on publication of this work.

Limitations of proposed approach. There are two limitations we are aware of. First, lazy FPU abduction, at least on x64, means that the floating point virtualization must also process many instructions that have nothing to do with floating point. This can result in consequential overheads that are higher than those for baseline floating point virtualization. When we analyze overheads in detail later, we will describe where these overheads come from and what could be done to mitigate them.

The second important limitation is inherent to the concept. Virtualization can only show the effect of lower precision on the *current* application binary, not what the effect would be on a *future* application with numerical methods and implementations specifically geared to the lower precision. If the application fails to run successfully with lower precision under virtualization, this does not mean it is impossible for it to do well under lower precision, just that the effort might be significant.

2 Floating Point Virtualization and FPVM

It is important to have a baseline understanding of how floating point virtualization and FPVM function.

Floating point virtualization seeks to extend the hardware floating point arithmetic environment available to an existing application binary without requiring modifications to the application binary. The architectural instructions embedded in the binary appear to execute as normal, but instead are selectively emulated

[2]The latter leverages the x64's instruction single-stepping mode, but breakpoint instruction patching would serve the same purpose on other architectures.

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. The result is that floating point virtualization has a transparency that is similar to that of a virtual machine monitor (VMM).

Our specific target system in this paper is the previously published and publicly available FPVM system [13, 16, 38]. FPVM has a well-defined interface to the alternative arithmetic system, which allows different choices to be compiled in. Here we use our new the "Teeny" system (§5).

User-level virtualization. Despite its analogous operation to a VMM, FPVM is designed to operate at user-level. This makes porting alternative arithmetic systems considerably simpler. Furthermore, typical architectures (e.g., x64, ARM, RISC-V) do not privilege the floating-point state or instructions, and FPVM has no protection goals of its own, so higher privilege is unnecessary. Finally, introspecting on the running process is considerably easier and faster in user space.

FPVM is an `LD_PRELOAD` library, meaning it is loaded early in the dynamic link order during the target process's startup so that it can establish the virtualization before any application code runs. This enables FPVM to override or modify the behavior of later-loaded shared libraries. FPVM's constructors allow it to initialize well before the `main()` of the process begins to do its own initialization and to follow `fork()`s and `exec()`s. Thread creation is also intercepted so that FPVM can create and maintain an execution context for each thread. For the purposes of this paper, FPVM 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 re-associate the execution context with the thread producing the signal. Virtualization operates on a per-thread basis, and is explained below.

NaN-boxing. An important difference between FPVM and traditional virtualization is that FPVM needs to use the alternative arithmetic system's representation of numbers without changing how the application's instructions themselves use them. FPVM does this through *NaN-boxing* [7, 39]. 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 consumes that value the hardware will trap to FPVM, allowing the operation to be performed in the chosen alternative arithmetic system.

Whose NaN is it? 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, FPVM extracts the pointer from the NaN and checks to see if its allocator remembers it. NaN handling is dramatically different in the present work.

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, resulting in a `SIGFPE` being delivered to FPVM.

Instruction Decoding and Emulation. In processing a `SIGFPE` signal, FPVM decodes the faulting instruction (using a decode cache to amortize this cost), binds its operands, and then emulates the instruction using the alternative arithmetic system. Decoding leverages the Capstone disassembler [1] and creates an FPVM-specific, architecture-independent representation that allows for portability to other architectures.

Garbage collection. Because FPVM allocates and encodes pointers to heap objects (the alternative arithmetic system's values) in (often temporary) NaNs, garbage can readily result, and it must be collected to avoid an explosion of memory usage. 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 traditional conservative mark-and-sweep garbage collection in which the objects managed by the collector never contain pointers themselves.

Correctness instrumentation. Unfortunately, x64 floating point hardware is not entirely virtualizable due to the previously mentioned admixture of floating point and non-floating point instructions that use the same state. Value flow through memory is also a concern for correctness. FPVM uses profiling and live patching to quickly return control to FPVM at instructions that may flow floating point values to non-floating point contexts. FPVM then does demotions of the values if needed. The present work reuses this functionality without changes.

3 Challenges and Opportunities

Independent of the specific architecture and alternative arithmetic system, tiny numbers simultaneously create new challenges for floating point virtualization but also simplify other aspects of it.

Challenge: naive use of native precision results in missed traps. Existing techniques for floating point virtualization rely on configuring the FPU to trap when a rounding or error exception occurs. Instructions whose result can be exactly represented by the hardware's native floating point format will not trap and will execute without any opportunity for the virtual machine to alter their result. To emulate a *less* precise format, *every* floating point instruction must be made to unconditionally trap or else they will be invisible to the virtualization framework.

In the present work, we introduce lazy FPU abduction (§4) as the mechanism to have all floating point instructions trap, but the hardware will also trap on a range of non-floating point instructions. We then filter this stream of traps, emulating only the floating point instructions, and single-stepping over the others.

Challenge: trap volume and overhead. Having every floating point instruction trap means that the trap volume may be much larger than when using higher-than-native precision, where precise hardware results do not cause traps. The higher trap volume implies

higher overhead for virtualization, though it is possible that techniques that amortize trap costs over multiple instructions may become more effective. Software and hardware techniques that reduce fundamental trap costs become more important.

In the present work, we amortize trap cost by emulating multiple instructions per trap, extending FPVM's "sequence emulation" [38] capability. Previous work also shows how to accelerate trap delivery of all kinds through both kernel support [38] and hardware support [16]. The present work does not use these features, though it could.

Opportunity: simplified NaN boxing and differentiation. Given that all floating point instructions must trap, all floating point values can be under direct control of the virtualization framework—there need be no hybrid operation in which ordinary IEEE 754 values and NaN-boxed alternative arithmetic values co-exist. As a result, the problem of determining whether a NaN is a NaN-boxed value or a "true NaN" goes away. Furthermore, the point of a NaN box encoding—to cause a trap to happen when an alternative arithmetic value is consumed—goes away since all instructions trap anyway. In the extreme, the values need not even be NaN-boxed at all, but could be directly encoded.

Opportunity: value boxing to avoid garbage collection and allow for virtualizing smaller hardware floats. When the alternative arithmetic system has higher precision than the hardware, we must encode pointers to alternative arithmetic values into hardware NaNs. This produces a garbage collection problem and makes it difficult to extend virtualization to smaller hardware floating point formats. When the alternative arithmetic system has lower precision, its values can be directly encoded into NaN values, avoiding the need for garbage collection entirely. Furthermore, the alternative arithmetic values can be used in smaller hardware floating point formats as long as they "fit".

In the present work, we preserve the concept of NaN boxing, but value-box the alternative arithmetic values instead of pointers to them. No garbage collection is then needed.

Opportunity: simplified alternative arithmetic system. Many tiny number systems of current interest are essentially IEEE 754 with varying numbers of exponent and mantissa bits and performance-oriented simplifications such as elimination of subnormals and complex rounding modes. Consequently, they are likely to be implementable on top of IEEE 754 doubles, making them far less complex than higher precision alternative arithmetic systems such as MPFR.

In the present work, our "teeny" system (§5) is implemented in this manner, with an optional extension to enable exploring mixed-precision use cases.

4 Lazy FPU Abduction

We will now describe "Lazy FPU Abduction": the mechanism which allows all floating point operations in a program to be made visible to FPVM, and thus a key enabler for virtualizing low-precision formats.

4.1 Requirements for Virtualization

Implementing lower precision floating point emulation with FPVM requires certain features from the hardware and operating system. These features overlap with those required to support lazily saving and restoring FPU state on a context switch. Hence we have coined the term *Lazy FPU Abduction* to describe them.

Unconditional Floating Point Traps. As mentioned previously, it must be possible to configure the floating point hardware to raise a trap on *every* floating point instruction. This is analogous to the Popek-Goldberg requirements for a virtual machine monitor [34]. In their terminology, for floating point virtualization, an instruction is *sensitive* if it operates on floating point values. An instruction is *privileged* if it the processor can be configured so it unconditionally traps when executed. And if there are sensitive instructions which are not privileged, the system is not fully virtualizable.

Note that this only requires that the floating point instructions are *subset* of the instructions which trap. More instructions than necessary may trap into the virtual machine monitor. If this is the case, as we will see on x64, the hardware must support executing a single arbitrary instruction before trapping back to the virtual machine. This can be implemented via a dedicated single-step trap mode, or via a breakpoint instruction and dynamic patching at runtime. We will refer to this feature as "single stepping" regardless of how it is actually implemented. If the hardware does not support single stepping, the virtual machine may need to emulate a larger subset of the instruction set than only those dealing with floating point values.

If the hardware supports single stepping then when the virtual machine encounters an instruction it does not recognize it can disable traps, single step the faulting instruction, and re-enable traps. This creates the illusion that the set of trapping instructions is exactly the set of floating point instructions. However if a large number of instructions must be single stepped over, we would expect a substantial overhead to be introduced.

Userspace Control. One of the major benefits of FPVM is that the virtual machine is a userspace shared library, loaded alongside the executable it is virtualizing. This greatly decreases the effort which must be put into adding new alternative arithmetic systems to FPVM because there is not need to port code into the kernel. However this means the virtual machine is running in userspace, and cannot access any privileged state. However FPVM will often need to be able to execute floating point instructions internally, without trapping back into itself. For example, when a trap occurs, often the native floating point hardware will be used to emulate the alternative arithmetic system replacing it.

Thus userspace must able to mask abduction when entering FPVM and unmask abduction when resuming execution of the main application. Ideally this would happen via direct access to the hardware, masking and unmasking abduction via a userspace accessible control register such as `mxcsr` which currently control the floating point rounding mode and trapping behavior for precision related exceptions on x64. On an unmodified version of x64, abduction requires using features initially designed to support lazy FPU save/restore, which means they are inaccessible from userspace.

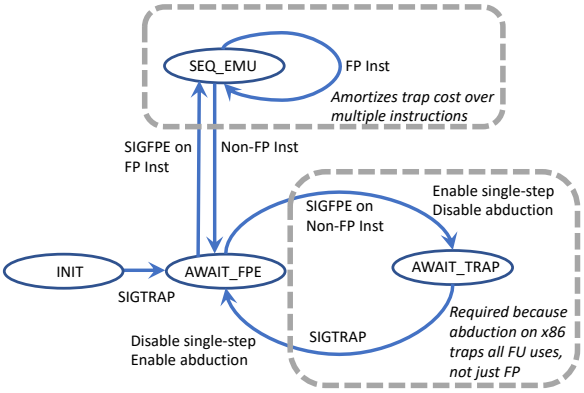


Figure 1: FPVM state machine as extended to support tiny alternative arithmetic systems. Because lazy FPU abduction captures all instructions that touch floating point state, not just floating point instructions, single-stepping is used to allow non-floating instructions uses to execute as normal, while floating point instructions are trapped and emulated. Sequence emulation is used to amortize the cost of the latter traps.

Thus a kernel module is required to provide the mask/unmask interface to userspace.

Process Local State Management. As a result of requiring userspace control of the trapping behavior, the kernel must also ensure that the current trapping behavior is saved and restored on a per-process basis. Otherwise when FPVM enables floating point traps on the current processor, any other process which utilizes floating point will begin receiving unhandled `SIGFPE` and crash. This includes any use of floating point within the *kernel*, which is rare.

To avoid this, the kernel must be modified to include the current trapping behavior in the saved state of a process, and it must add checks to any floating point operations done by the kernel itself, to ensure the userspace controlled value will not cause the kernel to trap.

4.2 Integration with FPVM

Figure 1 illustrates how FPVM's state machine has been extended to support virtualization with tiny numbers. The original state machine comprises only the INIT, AWAIT_FPE, and SEQ_EMU states. The AWAIT_TRAP state has been added, and the processing in the SEQ_EMU state has been expanded. Whenever FPVM itself is running, lazy FPU abduction is disabled. In the following enablement and disablement of abduction (and single-stepping) occurs when returning from FPVM to the user code.

In the AWAIT_FPE state, lazy FPU abduction is enabled so that all uses of the FPR state will trap into FPVM as a SIGFPE. If the faulting instruction is not a floating point instruction, the new AWAIT_TRAP state is entered, abduction is disabled, single-stepping mode is enabled, and the instruction is restarted. This allows the instruction to complete and the next instruction generates a SIGTRAP. The SIGTRAP results in a transition back to AWAIT_FPE along with enabling abduction and disabling single-stepping. At this point, the machine is once again looking for a faulting instruction. If while in AWAIT_FPE a SIGFPE occurs on a floating point instruction, FPVM transitions to SEQ_EMU and attempts to emulate as long of a sequence of instructions as is possible. In baseline FPVM, the sequence terminates when an unemulatable instruction occurs or an emulatable instruction that does not use any NaN-boxed operands occurs. In the present work, we eliminate the latter stopping condition because hybrid values do not exist—all emulatable instructions are or should be using the alternative arithmetic system's tiny values.

4.3 Kernel Module Implementation

In order to realize lazy FPU abduction in FPVM we implemented a Linux kernel module to allow access to the privileged features of the hardware needed. The module specifically targets Linux 5.15.0 though the functions the module needs access to are stable across a range of versions.

Per-Process Task Switched Bit. `x64` provides the ability to lazily save and restore floating point state via the "task switched" (`ts`) bit in the `cr0` control register. When the bit is set, any attempts by the processor to access a floating point register (`%xmm*,%ymm*,%zmm*`) will cause a "Device Not Available" (#NM) exception into the kernel. Because all floating point instructions operate on these registers, by setting `cr0.ts` the hardware will unconditionally trap on floating point instructions. Though because these registers can contain both floating point and integer values, a large number of non-floating point instructions will also fault. To deal with these the "Trap Flag" (`TF`) can be used to single step over the instruction with `cr0.ts` set to zero.

The `cr0` register is privileged and treated as global state within the kernel. Changing its value will change the current trapping behavior for *all* processes. The primary feature of the kernel module is to allow setting the value of the `cr0.ts` bit on a per-process basis. This effectively localizes the bit and allows FPVM-enabled processes using tiny number systems to coexist alongside other processes in the system which expect the bit will always be cleared.

When a process registers itself with the kernel, a preempt notifier [12] is attached which tracks the requested value of `cr0.ts` for the process and sets the privileged `cr0` register whenever control of the CPU is given to or taken away from the process. At any point the process may request that the module change its local `cr0.ts` by issuing a `write` to a sysfs file presented by the module. This requires a trip into the kernel, which can take on the order of a thousand cycles.

Lazy FPU Removal. In June 2018, a speculative execution side channel attack (`CVE-2018-3665`) was discovered which leverages lazy FPU saving [28]. In response, the Linux kernel developers disabled the feature by default. And since Linux version 4.10 the ability to save/restore FPU state using the `cr0.ts` bit has been completely removed from the kernel [29]. This has left the `cr0.ts` bit unused within the kernel, and so our dynamic module can assert control of the bit without causing issues with the core kernel.

#NM Hijacking. Since Linux now enforces eagerly saving/restoring FPU state the exception handler for the #NM exception has been drastically simplified. By default, the kernel now responds

to the exception by clearing `cr0.ts` and returning to the faulting process without delivering a signal. Our module utilizes kernel live-patching [11] to insert code which checks if the faulting process has registered itself, and then delivers a `SIGFPE` which can be caught by FPVM which begins emulation of the instruction.

The handler itself is a low level routine placed inside the interrupt descriptor table (IDT). This disallows patching/hooking the handler using any of the standard techniques from an external kernel module[3]. Luckily, there is an errata check at the beginning of the handler which can be patched using the standard Linux live-patching API. Without this specific Pentium Pro errata, implementing these changes as a dynamic kernel module would be nearly impossible.

Signal Masking. In addition to tracking the requested value of `cr0.ts`, the kernel module must also track whether the process is currently servicing a previously delivered `SIGFPE` and mask the hardware value of `cr0.ts` if it is still handling the previous fault. This way alternative arithmetic system can use floating point to implement the emulated operations, without itself trapping and infinitely recursing. To avoid faulting again, the module tracks and additional bit per process, which is set when a `SIGFPE` is delivered, and cleared when the `sigreturn` system call is invoked. The process is given the ability to set this bit manually via a `write` system call to a sysfs file. This allows floating point state to be manipulated inside signals such as `SIGTRAP` which is needed by FPVM but not delivered by the kernel module.

Kernel Use of Floating Point. Generally, Linux will avoid using the FPU when running inside the kernel. Though in special circumstances the kernel will need access to the floating point registers, for example: when saving the floating point state of a process. To handle these cases, the kernel module applies `kretprobes` which clear the `cr0.ts` bit on entry to a function and reset it on return, to a list of functions which will need access to the floating point state. This approach is extremely flexible, and has worked well for our evaluations, but has a few downsides. First, `kretprobe` is "best-effort", if demand on a specific function is high enough, it is possible that the callbacks will not be invoked. Second, the current list of functions which need to be probed may be incomplete. In either case, the kernel will attempt to access floating point state with the `cr0.ts` bit set, and cause an exception. To handle this, the kernel module checks if the fault occurred while running in the kernel, logs a warning, and clears the `cr0.ts` bit. This allows the kernel to survive the fault, but may cause the application to run without trapping for the rest of the current timeslice. However with a reasonable number of probes (one per CPU) this event is exceedingly rare. In addition this "leak" is fully detectable, and so if it did occur, the user can be notified and then retry the program.

5 "Teeny" Arithmetic with Value Boxing

We virtualize hardware double precision floating point to any variant of lower precision IEEE 754 floating point using value boxing combined with an alternative arithmetic system we refer to as "teeny". Figure 2 illustrates how value boxing works, as well as the format of a teeny number.

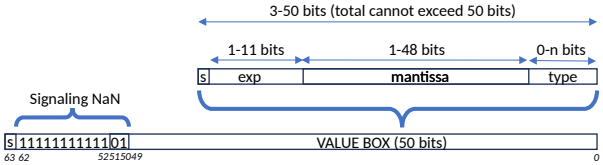


Figure 2: Value boxing an alternative arithmetic value inside of a double precision signaling NaN and format of a teeny number. Unlike traditional NaN-boxing of pointers to alternative arithmetic values in FPVM, value-boxing directly stores the value (up to 50 bits) in the NaN. Using this space, the teeny arithmetic system can encode floating point numbers with a wide range of exponent and mantissa bit-widths.

The hardware's 64-bit IEEE-754 floating point format consists of a 52 bit mantissa, 11 bit exponent, and 1 sign bit. To encode a NaN, all 11 exponent bits must be 1's, and the mantissa must be non-zero. If the most significant bit of the mantissa is a 0, the NaN will be "signaling", while if it is 1, the NaN will be "quiet".[4] By fixing the two most significant bits of the NaN payload to 01 (signaling NaN), we can use the remaining 50 bits of the NaN's "payload" [20–22] to correctly encode any data, including 50 zero bits.

In baseline FPVM the 50 payload bits are used to stash a pointer to a structure which encodes a value in the alternative arithmetic system. When encoding an alternative arithmetic value that requires 50 bits or fewer, the value can be encoded directly in the payload bits, or "value boxed".

The "teeny" alternative arithmetic system uses value boxing to encode a sign bit, biased exponent, and mantissa as in a IEEE-754 floating point number. It allows distributing the 49 available bits [5] between the exponent, mantissa, and an optional "type" field used to support mixed-precision (§5.1) however the user sees fit. We limit the teeny exponent to 11 bits so that the maximum range of a teeny stays within the range of a 64-bit double. Indeed, teeny numbers nest within the doubles, meaning that every teeny has an exact double equivalent. This supports encodings ranging from using all 50 bits to encode a 11 bit exponent and 38 bit mantissa, down to a 3 bit format, only allowing a single bit for each of the sign, exponent, and mantissa.

As illustrated in Figure 3, operations on value-boxed teeny numbers can be readily implemented by unboxing the numbers, decoding the teeny numbers (which exactly converts them to 64-bit doubles), performing the operation on the doubles, encoding the result into a teeny number (where the rounding mode is selectable), and then value-boxing the teeny. It is important to note that *any* alternative arithmetic system with 50 bit or smaller values could readily be incorporated into this model, simply by handling the decoding, operation, and encoding steps differently, as noted in the figure. For example, > 11 bit exponent fields are certainly possible, just not through re-using double precision arithmetic as teeny does.

[3] In theory the address of the IDT could be determined and the function overwritten. But this would make *removing* the kernel module once it was installed much more difficult.

[4] The distinction between signaling and quiet NaNs is not important to understand here other than to note that signaling NaNs will cause floating point exceptions and traps in a wider range of situations, which why we use signaling NaNs.

[5] One bit must be dedicated to the sign.

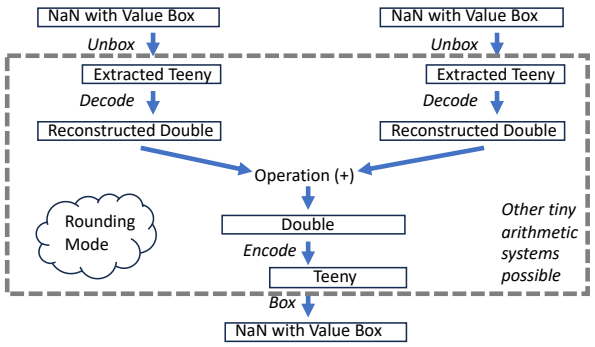


Figure 3: Processing flow for an operation on teeny numbers. Input teeny numbers are unboxed and then decoded to double precision. The operation is executed in double precision and the result is then encoded into a teeny that is then value boxed. Decoding is precise since teeny numbers nest in double precision numbers, while encoding depends on a rounding mode.

Why NaN-box? An observant reader may have noticed that because FPU abduction allows trapping on every floating point operation, and because teeny values are able to nest inside double values, NaN-boxing is not strictly necessary for virtualization. The virtualization system could instead leave all values as valid doubles, and simply round the result of every floating point instruction to the nearest value representable by the lower precision format.[6]

While this is a potential optimization, there are still a number of reasons to continue NaN-boxing all values tracked by FPVM.

(1) NaN-boxed values will continue to trap and be virtualized properly even if FPU abduction is disabled. This opens the door to a form of "lossy" virtualization where abduction is intermittently disabled to achieve higher performance at the expense of potentially leaking some higher precision into the calculation.
(2) NaN-boxing allows encoding additional data not found in a standard IEEE 754 floating point value. This includes the optional "type" field which allows mixed precision use cases for teeny values (§5.1).
(3) A NaN-boxed teeny can potentially exist alongside even *higher* precision, pointer based, formats which can be virtualized by FPVM (GNU MPFR, LibBF, Posits, etc.).
(4) The act of boxing/unboxing a value is a single comparison and a small number of bitwise instructions, which as we will see in §6, is a tiny cost compared to the overhead of trapping at all.
(5) The implementation of FPVM is tied to the assumption that all values will be NaN-boxed. It would not be impossible to change this, however the limited benefits of doing so would not seem worth the likely significant engineering effort.

[6]Considering Figure 3, this would mean that the path from"NaN with Value Box" to "Reconstructed Double" would simply be replaced with "Rounded Double" and the path from "Double" to "NaN with Value Box" on the output would be replaced with "Rounded Double". The *encoding* step would then round the output Double of the *operation* to the limits implied by the selected Teeny number representation, producing a "Rounded Double". One advantage of this model is that it could immediately support any mantissa and exponent bitwidths up to those of a `double`.

5.1 Mixed-Precision

Programs which could benefit from selective use of low precision, are often unable to benefit when the precision is reduced across all values. Thus when considering whether or not to port an application to a lower-precision format, being able to explore mixed-precision options is vital.

When applying mixed-precision to an existing program, there must be some means of choosing which precision to use for the various values generated by a program. The most straightforward method is to choose a single precision per-instruction, then have all values produced by that instruction use this precision. We will refer to this as specifying precision "instruction-by-instruction". This maps well onto physical hardware but can limit the ability of the programmer to explore how a program reacts to a lower precision.

An alternative method could use the precisions of the input values to an operation to decide the output precision. The same instruction could then produce values of varying precisions when the precision of its inputs varies. We will refer to this as specifying precision "value-by-value", because each value must encode its own precision.

Compiler based approaches naturally tend to specify how mixed precision is applied instruction-by-instruction, as encoding precision value-by-value would require more complicated program analysis and runtime support. We observe that a virtualization based approach is more amenable to specifying the precision value-by-value, because techniques such as NaN-boxing allow easily adding additional information to each value generated. To demonstrate this, we have implemented a mixed-precision mode of operation for the teeny format which can capture both instruction-by-instruction and value-by-value semantics.

We do so by allowing some number of bits per value to be dedicated towards encoding the precision. Because any bits used to encode the precision must come from the same set of 50 bits used to encode the sign, exponent, and mantissa, we encode the precision indirectly via an integer "type" field. When unpacking a value-boxed teeny FPVM first reads the type and uses it to index a user provided array of valid precisions. This tells FPVM the bitwidths of the other fields, allowing the value to be unpacked into a double. This indirection also minimizes the simple case of having a "high" and "low" precision, in which only one bit is required.

To allow the user to control how values of different precisions interact, we introduce the notion of *conversion rules*. These are a user-provided mapping from input precisions to output precisions which FPVM uses when virtualizing an operation. A simple set of such rules can be seen in Figure 4. The first section of the file defines two different precisions a "low" precision 16-bit `bfloat16` and a "high" precision 32-bit `float32`. The default conversion rules are then set so that if the inputs to an operation are of the same precision, then the result uses that precision. However if the inputs have different precisions, then the output should utilize the lower precision format. This demonstrates how to encode a simple value-by-value mixed-precision system.

```
type low = 8:7 ; bfloat16
type high = 8:23 ; float32

; By default values of the same precision
; produce values of their own precision,
low low   -> low
high high -> high

; If inputs to an operation have
; different precisions, use the smaller
; precision for the result.
; (value-by-value)
low high -> low

; Force a range of instructions to
; use solely float32 values
; (instruction-by-instruction)
region 0x401610 0x401756 {
  low low   -> high
  low high  -> high
  high high -> high
}

; If a bfloat16 would overflow or underflow,
; convert to a float32 value instead.
; FPVM can then log exactly which instructions
; causes such conversions.
underflow low -> high
overflow  low -> high
```

Figure 4: An example set of mixed-precision conversion rules. It utilizes value-by-value semantics for the majority of the program and instruction-by-instruction within a single function. FPVM will check an environment variable to find a file similar to this during program start up.

Ranges of instruction addresses can be associated with custom sets of conversion rules. This allows treating regions of the same program differently. In Figure 4 the function residing in the address range `0x401610` to `0x401756` has a set of such custom rules. Instead of using the value-by-value conversion rules which the rest of the program uses, this function maps all input precisions to the same output precision (float32). This effectively implements instruction-by-instruction semantics as the instructions produce the same output precision independently of the input values.

Because every operation in FPVM is either unary or binary, an arbitrary set of conversion rules can be stored in memory and efficiently queried as a 2-dimensional lookup table with size proportional to the square of the number of precisions requested. Thus assuming a reasonably small number of precisions being used, the overhead of setting unique conversion rules for many individual ranges of instructions can be quite low.

We have also extended these rules to allow limited conversions if values such as infinity or zero would be produced due to a rounding event. This allows more complicated behavior such converting a value to a higher precision *only* if the lower precision would have caused that value to overflow or underflow. FPVM can then be configured to log exactly which instructions caused such a conversion, immediately pointing the programmer towards portions of the program which may need to be modified when porting to a lower precision. With an purely instruction-by-instruction approach, the first event of overflow or underflow is likely to lead to erroneous behavior by generating a NaN or infinity. These values can then propagate throughout the program leading to far more false positives when trying to determine which portions of the program are truly incompatible with the lower precision.

FPVM loads these conversion rules at program start up by reading them from a file such as Figure 4. This means a large number of different mixed-precision configurations can be rapidly tested without recompiling the program or FPVM.

Examples. To see how these conversion rules can be useful, consider specifying two precisions: a low-precision "A" (the default) and a high-precision "B", and then running a program with the following conversion rules:

$$\{A, A\} \mapsto A$$
$$\{A, B\} \mapsto A$$
$$\{B, B\} \mapsto A$$
$$A \mapsto B \text{ on overflow}$$
$$A \mapsto B \text{ on underflow}$$

The program will then run with the low precision exclusively until the lower precision would cause an overflow to infinity or underflow to zero. If this occurs, these conversion rules instead direct FPVM to convert the low precision value to a higher precision for a single instruction. FPVM can then log exactly which instructions generated higher precision values, pointing the programmer towards potentially problematic portions of the program.

We can also consider a slightly different set of conversion rules:

$$\{A, A\} \mapsto A$$
$$\{A, B\} \mapsto B$$
$$\{B, B\} \mapsto B$$
$$A \mapsto B \text{ on overflow}$$
$$A \mapsto B \text{ on underflow}$$

These map to the higher precision if *any* of the inputs are higher precision. This effectively makes the higher precision "sticky" and causes it to propagate to any values which depend on the result of the instruction which had an exceptional event. This provides a simple mechanism for the programmer to see to which program values could be directly affected by overflow/underflow from the loss of precision, simply by examining which values are high precision at program termination.

It is important to note that value-by-value conversion rules are meant to enable more flexible exploration/debugging of an existing program under mixed-precision with minimal effort. However they still permit strict adherence to instruction-by-instruction precision by mapping the resulting precision of any given instruction to be

a constant. Thus providing a mode of operation which can mimic real mixed-precision hardware much more closely.

6 Evaluation

Our evaluation comes in two parts. The first demonstrates the effect of low-precision formats on a range of benchmarks in order to help validate the correct emulation of these formats. The second measures the performance overheads introduced by FPVM and Lazy FPU Abduction, including a breakdown of individual costs introduced by each step of emulating a floating point instruction.

6.1 Testbed

All testing was performed on a Dell R6515 containing a single AMD EPYC 7443P 24-Core processor, a single NUMA zone, and 64 GB of RAM. It runs Ubuntu 22.0.4.5 LTS with the 5.15.0 Linux kernel, and supports the SSE4.2 and AVX2 floating point instruction sets. This is an identical configuration to that described in [38] and thus allows for close comparison of behavior between virtualizing using the fastest "traditional" alternative arithmetic system (NaN pointer boxed complete 64 bit floating point values) and this paper's NaN value-boxed "teeny" numbers (down to 3 bits).

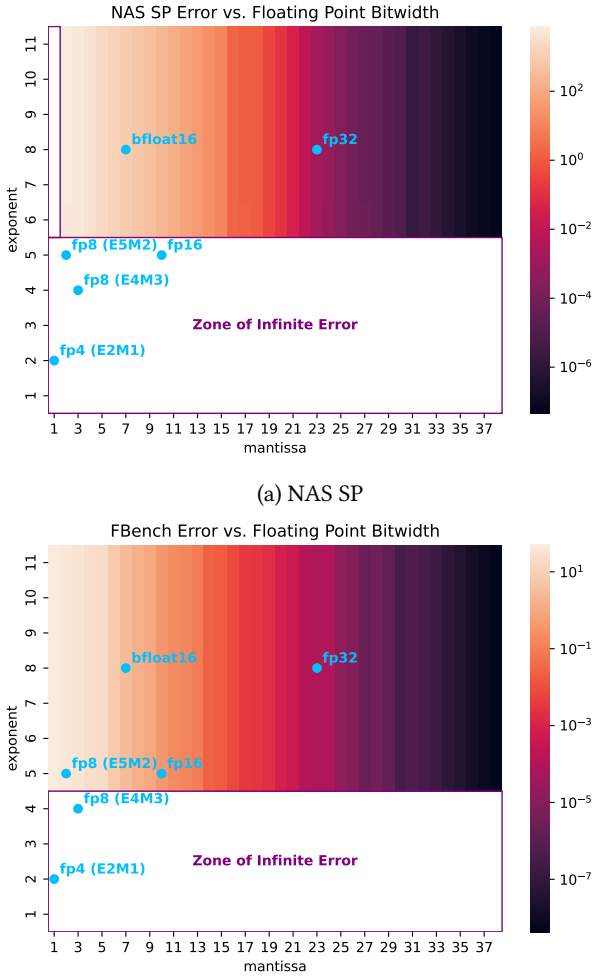
To allow comparison to previous papers involving FPVM [13, 38], we evaluate our system using the FBench floating point benchmark suite [36], the FFBench fast Fourier transform benchmark [37], a Lorenz system simulator, and Enzo [6], an astrophysics and hydrodynamics simulator. We also incorporated a wide selection of the NAS benchmark suite (BT, SP, IS, FT, CG) [3, 23, 33].

6.2 Validation and use case

To validate our system, we ran FPVM with "teeny" floats on a subset of benchmarks (NAS SP, FBench, and Lorenz) using every supported combination of bitwidths for the exponent and mantissa and observed the effects on error. Error is directly drawn from NAS SP's output. For FBench, the program's numerous error outputs are used to generate an error vector for which we compute the mean squared error (MSE). For Lorenz, the final 3D position is differenced with the double precision result and we report the MSE.

Note the fact that seeing "errors" when lowering precision during validation is not a cause for concern. In fact, it would be alarming if a benchmark did *not* report any errors. All of the benchmarks tested were designed with 64-bit double precision floating point values in mind, and any error bounds associated with them reflect that fact. By running these applications with a smaller floating point representation we reduce the total range of representable values and increase the size of gaps between them. Overflow to infinity and underflow to zero then occur at smaller and larger values respectively and every rounding event introduces a larger amount of error than with a larger format. This effect is most pronounced when the exponent is restricted.

Note that with 11 bit exponents, and 38 bit mantissas, the "teeny" format approaches the native doubles (11 bit exponent, 52 bit mantissas) these benchmarks use. As our configuration approaches that limit, error is minimized for what we refer to as convergent systems, as expected. For chaotic systems, this is not the case, but this is also as expected, as we discuss below.



(a) NAS SP

(b) FBench

Figure 5: Reported error of NAS SP and FBench (convergent systems) while varying bitwidth exponent and mantissa. FPVM behaves as expected.

Convergent Systems. NAS SP attempts to solve a set of complex systems of equations numerically. Because there exists an analytic solution to these equations, we expect the approximation calculated by SP to converge on the true value. FBench implements a raytracing algorithm designed to stress the implementation of trigonometric functions. FBench also converges on a solution, with the double precision solution being known exactly.

Figure 5 illustrates the result of our sweeps of the number system format under NAS SP and FBench. Also noted on the figure are where a range of lower-precision alternative number systems (down to 4 bits) fall in the spectrum. Of course, a use case for FPVM with tiny numbers is simply to see which formats are feasible for an application and how its error will react to lower precision.

Because both SP and FBench approach a fixed solution, we'd expect the error of running with a smaller number format to vary

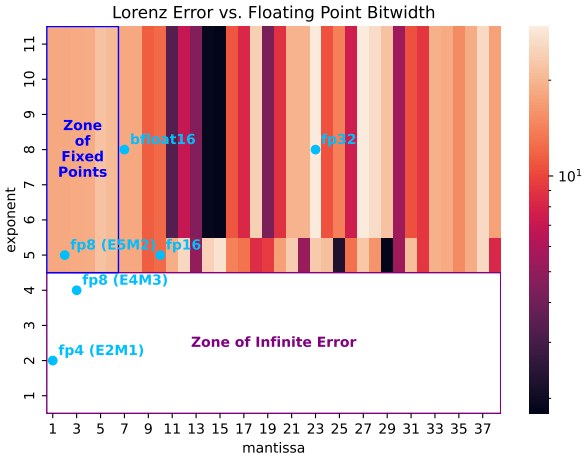


Figure 6: Error of Lorenz system simulation (a chaotic system) while varying bitwidth exponent and mantissa. Error is relative to the final system state calculated using double precision values. FPVM behaves as expected.

proportionately with the error introduced by rounding this fixed solution to fit within the representation. If the mantissa is m bits wide, and we hold the exponent constant, then the absolute difference between two consecutive values, and thus the error introduced by a rounding event, is proportionate to 2^{-m}. This is exactly what we see in the SP and FBench figures when we hold the exponent constant (taking a horizontal slice through the graph). With a sufficiently large exponent bitwidth, we see an exponential increase in the error as we decrease the number of mantissa bits. The plot expresses error values logarithmically, so an exponential function appears as a linear gradient.

Chaotic Systems. Unlike SP and FBench, Lorenz does not converge to a specific value when errors are introduced. It is a chaotic dynamic system, which means a very small perturbation to the input values (or error introduced during computation) will result in a large changes in the final result [32]. As such, the error introduced by FPVM's tiny numbers is not correlated with the width of the mantissa, instead appearing as random noise, as we would expect.

With small enough mantissas, Lorenz exhibits an additional behavior: fixed points/cycles. These occur when within some finite number of time steps the simulation returns to the same point more than once. Because the application is fully deterministic, this causes the program to repeat the same set of points cyclically. We believe that this behavior can be explained by the reduction in total number of possible values. For illustration consider a hypothetical 13-bit format "`E11M1`" that has the same number of exponent bits as a double precision value and so covers nearly the same range of values. However there are only $2^{13} = 8,192$ possible values. Therefore we would expect a system to encounter a cycle within that many time steps. Fully characterizing this behavior is outside of the scope of this work, though it should be noted that cycles and fixed points are well studied in the field of dynamical systems such as the Lorenz attractor [10, 32].

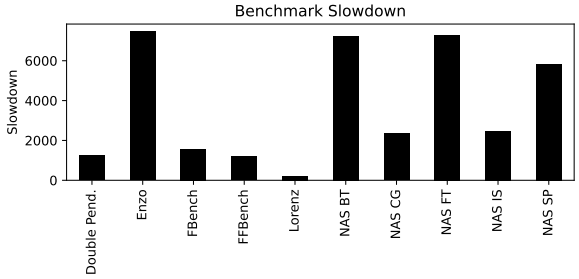


Figure 7: Slowdown in realtime of various benchmarks versus baseline (no virtualization).

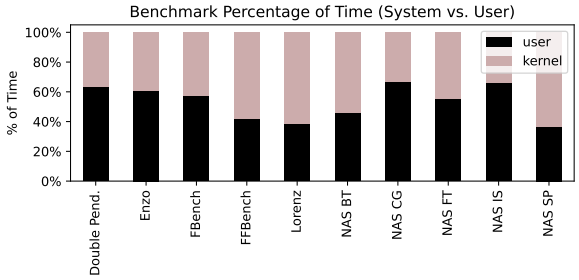


Figure 8: Measured proportion of time spent in user and system with FPVM and lazy FPU abduction.

Mantissa is All You Need. We observe here that (and the effect appears the same in the other benchmarks) once the exponent bitwidth is large enough, increasing it further has very little effect on the error, *even for a chaotic system.* This reflects the fact that increasing the number of exponent bits leads to more values near zero and infinity. However, it will *not* lead to increased resolution between two non-zero finite values. If the exponent width is sufficient to represent the largest and smallest values an application will ever require, then increasing the exponent width further will not lead to more precise results. In contrast, if the exponent is *not* wide enough, then an overflow to infinity, or an underflow to zero[7], will occur and propagate, leading to an infinite or undefined error.

This implies there is an ideal exponent bitwidth for any specific application, which minimizes the exponent size while still covering the full range of values required. We have demonstrated that FPVM can be used to determine this minimum on arbitrary programs and workloads *empirically.* In addition to using this information (and sweeps like those underlying Figures 5–6) to choose from among available alternative number formats, it may also be helpful in designing accelerators for specific applications (e.g., FPGA datapaths).

6.3 Performance Overhead

As seen in Figure 7, for most benchmarks tested, our prototype produces slowdowns on the order of a thousand compared to running without virtualization. This is despite utilizing some of the

[7]Which can result in Infinities or NaN's via new division-by-zero errors.

optimizations introduced in [38], notably sequence emulation, and is about 2–10× that shown in that previous work.[8] This change reflects the fact that we must now trap on *every* instruction that touches floating point state (e.g. `xmm` registers), while the previous work only needs to trap on rounding events and use of NaN boxed values. The increased number of traps is reflected in Figure 8, which shows that often up to *half* of the benchmark runtime is being spent in the kernel.

Figure 9 breaks down the amortized cost, in cycles, per useful emulated instruction into their constituent parts. *hw + kern* constitute the signal delivery cost noted above, while *set_ts + clear_ts + mark_in_signal* constitute the communication cost with our kernel module. Another important cost unique to this work is *single_step_kern*, which reflects the hardware and software costs involved in single-stepping over useless instructions. The remaining costs reflect core FPVM processing and are comparable to the other work: *decache* (decoded instruction cache), *decode* (instruction decode), *bind* (operand binding), *emul* (instruction emulation and alternative arithmetic system), and *fcall* (correctness handling for foreign calls) and *corr* (correctness handling for NaN-leaks). Garbage collection cost is zero compared to previous work, and is elided here.

Signal Delivery. Our system contributes to the time spent in the kernel through two different mechanisms: signal delivery and communication with our kernel module. The time spent in the kernel for signal delivery is by far the more important of the two mechanisms. In fact, signal delivery is the largest cost per useful emulated instruction we observe for every benchmark tested.

The total amortized cost of signal delivery per useful emulated instruction (*hw+kern* in Figure 9) is only 2–5x that of FPVM without lazy FPU abduction. This is surprising, as without abduction, only instructions which are imprecise or operate on NaN boxed values will trap.

One contributing factor to this surprisingly low expansion of the overhead is that when lazy FPU abduction is enabled, FPVM will extend sequence emulation to the maximum length possible. By default, sequence emulation will only continue emulation if the inputs to an instruction are already NaN boxed, otherwise it will terminate emulation early. With lazy FPU abduction, we can be sure instructions will *always* trap regardless of their inputs. And so if a trap is inevitable, sequences should be extended as to avoid as many such traps as possible. Thus while we must handle many more instructions, we are also processing more instructions (longer sequences) per signal.

While signal delivery is the greatest cost per instruction for every benchmark in Figure 9, the importance of other factors varies much more across benchmarks. For benchmarks which rely on a relatively large number of vectorizable integer operations such as double_pendulum or NAS IS (integer sort) the cost of single stepping over non-floating point instructions which trap due to `cr0.ts` is second only to signal delivery in terms of cost. But for benchmarks which make very little use of integer operations (or are simply unable to be vectorized) such as FFbench the small number of single step traps are almost entirely insignificant. This makes determining the order of importance for the remaining overheads unclear, so we will discuss the remaining significant overheads in no particular order.

Instruction Single Stepping. Each time an instruction is encountered that traps but cannot be emulated by FPVM, such as integer vector instructions, we must pay the approximately 6,000 cycle cost of signal delivery unnecessarily. In fact the cost in this case is actually > 2× the cost of an emulatable instruction trapping: FPVM must mask traps, enable single stepping or patch in a temporary breakpoint instruction, receive a `SIGTRAP` with roughly the same signal delivery cost, and then unmask traps again. In total this raises the cost of a single unemulatable instruction much closer to 14,000 cycles.

Despite the high cost of a stepping over a single unemulatable instruction, the measured amortized overhead of dealing with such instructions (*single_step_hw + single_step_kern*) tends to be far less. At first it would seem sequence emulation, which helped amortize the cost of signal delivery for floating point instructions that trap, would not help alleviate the cost of non-floating point traps. However, FPVM has some support for emulating non-floating point instructions, such as basic vector `mov`'s. Originally this was intended to bridge gaps and build slightly longer sequences of emulated instructions. Here it means that many such `mov`'s and similar instructions are serviced during a single `SIGFPE`, avoiding single-stepping them.

Trap Short Circuiting Optimization. We can now observe the fact that for most benchmarks, the two largest overheads observed are tied to the high cost a single signal delivery from the kernel. The dominance of signal delivery in the cost of FPVM has been a long standing issue, and [38] implemented a system called "trap short-circuiting" to help alleviate it. This bypasses the standard path through the Linux kernel for `SIGFPE` delivery after a floating point exception (`#XF`), and results in an 8x reduction in overhead caused by signal delivery. Beyond engineering effort, we see no reason why this same technique cannot be applied to the `#NM` and `#DB` exceptions to see a similar reduction in overhead. Another technique [16] modifies the RISC-V architecture to entirely bypasses the kernel for such signal deliveries, and we believe it would also apply.

Instruction Emulation. While the total amortized cost of a single instruction is relatively low considering the drastically increased number of floating point traps, the amortized cost of emulation (*emul* in Figure 9) is relatively high, taking a few thousand cycles per emulated instruction for some benchmarks. This is especially surprising given how simple the implementation of the "teeny" number system is. It only takes ~100 cycles to unpack/pack a teeny to/from a double, and this is the limit for *emul*.

The fact that the emulation cost is higher than this limit reflects that *emul* also covers the cost of processing the instruction, not just running "teeny" operations. More importantly is that the cost of emulating non-FP instructions during sequence emulation (such as the `mov` instructions that would otherwise result in single stepping), is charged to *emul*. However, *emul* is amortized over the number of FP ("useful") instructions. This expansion of *emul* is well worth

[8] Note that the overhead includes that of the alternative arithmetic system. The previous work brought overheads to about 1.7–5.9× of that limit.

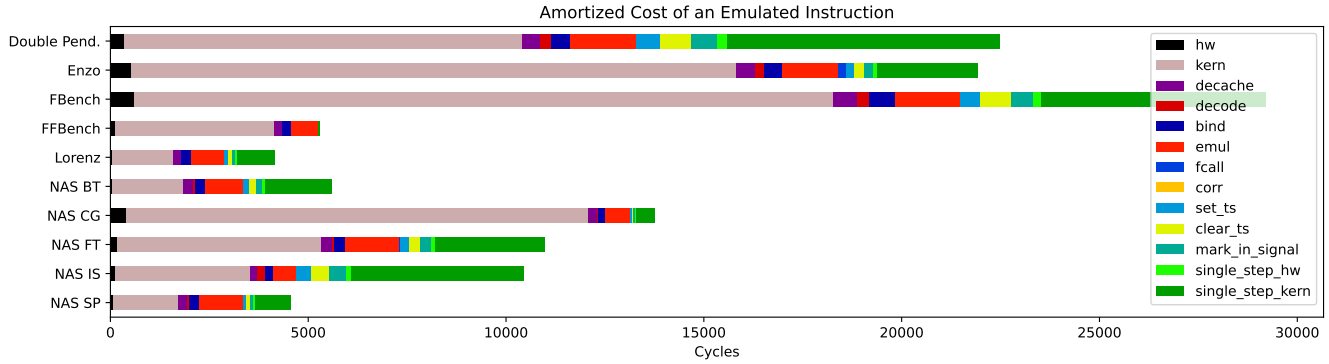


Figure 9: Amortized cost per useful emulated instruction, broken down to constituent parts.

Component	Lines C/ASM
Lazy FPU Abduction Kernel Module	957
FPVM Core (Total)	~510
Sequence emulation additions	~10
Single-stepping	~200
GC changes	~50
x64 kmod interaction	~200
General x64 changes	~50
Teeny Alternative Arithmetic System	1,190
Total	~2,657

Figure 10: Software engineering complexity as expressed in the number of new lines of C/ASM code (vast majority is C) needed to implement the concepts of this paper.

it since each non-FP instruction so handled avoids the giant 14,000 cycle cost of handling it via single-stepping.

Kernel Module Communication. We measure the cost of a single trip into the kernel to modify `cr0.ts` as 1018 cycles to set and 1139 cycles to clear. We also measured the cost of a trip into the kernel to notify the kernel module that the program entered a signal handler as 1101 cycles. All three of these operations are implemented as `write` system calls to sysfs files, with the only difference internally being that the signal notification writes to a memory address instead of actually changing the `cr0` register, which implies that the cost of modifying `cr0.ts` (while not measured directly) is not significantly different than the cost of writing to memory.

In a hypothetical system that exposed `cr0.ts` as an unprivileged flag, these costs (*set_ts* + *clear_ts* + *mark_in_signal* in Figure 9) could be avoided.

6.4 Engineering Complexity

Figure 10 illustrates the complexity of our implementation, breaking down between the kernel module (which enables lazy FPU abduction), changes to the FPVM core and x64 support to make use of those changes, to interact with the kernel module, and to otherwise implement the enhanced state machine of Figure 1, and the teeny alternative arithmetic system that it can drive. Adding tiny number support to the existing FPVM floating point virtualization system expands it by about 10%, with almost half of that expansion being the addition of the teeny alternative arithmetic system itself. The changes to the FPVM core (510 lines in total) comprise about 2% of the codebase size. These results suggest the techniques described in this work are likely to be tractable to apply in other tools.

7 Conclusions

We have shown how to extend floating point virtualization to arbitrarily low precisions. Floating point virtualization is an operating systems-level approach to plugging in an alternative arithmetic system under existing application binaries that is based on trap-and-emulate processing of individual floating point machine instructions. The lazy FPU abduction technique we have described makes it possible for all the traps needed to virtualize lower precisions to be generated by the hardware. We have also discussed other challenges and opportunities involved in low precision virtualization. Our prototype kernel/user implementation in the FPVM system for x64 is able to support any 50 bit and smaller number representation, and we provide an implementation that covers many common ones that nest cleanly in IEEE 754 doubles. Validation experiments and performance measurement were performed, driven by a range of benchmarks in binary form. Although our implementation is x64-specific, we believe that other architectures are likely to expose the necessary functionality (e.g., the `CPACR` and `misa` registers on ARM and RISC-V, respectively.)

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] Capstone: The ultimate disassembler, 2021.
[2] Arnold, M. G., Bailey, T. A., Cowles, J. R., and Cupal, J. J. Redundant logarithmic arithmetic. *IEEE Transactions on Computers 39*, 8 (Aug. 1990), 1077–1086.

[3] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinksi, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S. The nas parallel benchmarks (nas 1). Tech. Rep. RNR-94-007, NASA, March 1994.
[4] Bellard, F. Libbf: The tiny big float library. Available at https://bellard.org/libbf/, 2017.
[5] Boehm, H.-J. Towards an api for the real numbers. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2020).
[6] Bryan, G. L., Norman, M. L., O'Shea, B. W., Abel, T., Wise, J. H., Turk, M. J., Reynolds, D. R., Collins, D. C., Wang, P., Skillman, S. W., Smith, B., Harkness, R. P., Bordner, J., Kim, J.-h., Kuhlen, M., Xu, H., Goldbaum, N., Hummels, C., Kritsuk, A. G., Tasker, E., Skory, S., Simpson, C. M., Hahn, O., Oishi, J. S., So, G. C., Zhao, F., Cen, R., Li, Y., and The Enzo Collaboration. ENZO: An Adaptive Mesh Refinement Code for Astrophysics. *The Astrophysical Journal 211*, 2 (March 2014), 19.
[7] Cherkaev, A. The secret life of a nan. https://anniecherkaev.com/the-secret-life-of-nan, March 2018.
[8] Courbet, C. Nsan: A floating-point numerical sanitizer. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC)* (March 2021).
[9] Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. Qlora: efficient finetuning of quantized llms. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (NIPS 2023)* (2023).
[10] Devaney, R. *An Introduction to Chaotic Dynamical Systems*. 10 2021.
[11] Developers, L. K. Linux kernel documentation: Livepatch.
[12] Developers, L. K. Linux kernel documentation: preempt_notifier_register.
[13] Dinda, P., Wanninger, N., Ma, J., Bernat, A., Bernat, C., Ghosh, S., Kraemer, C., and Elmasry, Y. FPVM: Towards a floating point virtual machine. In *Proceedings of the 31st ACM Symposium on High-performance Parallel and Distributed Computing (HPDC)* (June 2022).
[14] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., and Zimmermann, P. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS) 33*, 2 (June 2007).
[15] Gustafson, J. *The End of Error: Unum Computing*. Chapman and Hall/CRC, 2015.
[16] Hallsby, K., Strand, L., Wanninger, N., Dhiantravan, N., and Dinda, P. Architecture-independent floating point spying and an architecture for floating point spying. Tech. Rep. NWU-CS-2025-27, Department of Computer Science, Northwestern University, July 2025.
[17] Hickey, T., Ju, Q., and Van Emden, M. H. Interval arithmetic: From principles to implementation. *Journal of the ACM 48*, 5 (Sept. 2001), 1038–1068.
[18] Hoerold, F., Ivanov, I. R., Dhruv, A., Moses, W. S., Dubey, A., Wahib, M., and Domke, J. Raptor: Practical numerical profiling of scientific applications. arXiv:2507.04647v2, September 2025. v1 is July, 2025; DOI is https://doi.org/10.48550/arXiv.2507.04647.
[19] Hoerold, F., Ivanov, I. R., Dhruv, A., Moses, W. S., Dubey, A., Wahib, M., and Domke, J. Raptor: Practical numerical profiling of scientific applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC 2025)* (November 2025). To Appear.
[20] IEEE Floating Point Working Group. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985* (1985).
[21] IEEE Floating Point Working Group. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70.
[22] IEEE Floating Point Working Group. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 2019), 1–84.
[23] Jin, H., Frumkin, M., and Yan, J. The open mp implementation of nas parallel benchmarks and its performance (nas 3). Tech. Rep. NAS-99-011, NASA, March 1999. OpenMP 3.0 version available at https://github.com/benchmark-subsetting/NPB3.0-omp-C.
[24] Jost, T., Durand, Y., Fabre, C., Cohen, A., and Pétrot, F. Vp float: First class treatment for variable precision floating point arithmetic. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)* (September 2020).
[25] Jost, T. T., Durand, Y., Fabre, C., Cohen, A., and Pérrot, F. Seamless compiler integration of variable precision floating-point arithmetic. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (February-March 2021).
[26] Kahan, W. 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)* (July 2016).
[27] Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D. T., Jammalamadaka, N., Huang, J., Yuen, H., Yang, J., Park, J., Heinecke, A., Georganas, E., Srinivasan, S., Kundu, A., Smelyanskiy, M., Kaul, B., and Kundu, P. D. A study of BFLOAT16 for deep learning training. arXiv preprint arXiv:1905.12322, May 2019.
[28] CVE-2018-3665. Available from RedHat, CVE-ID CVE-2018-3665., June 12 2018.
[29] x86/fpu: Finish excising 'eagerfpu'. Commit to the Linux kernel.
[30] Matula, D. W., and Kornerup, P. Finite precision rational arithmetic: Slash number systems. *IEEE Transactions on Computers C-34*, 1 (Jan 1985), 3–18.
[31] Micikevicius, P., Stosic, D., Burgess, N., Cornea, M., Dubey, P., Grisenthwaite, R., Ha, S., Heinecke, A., Judd, P., Kamalu, J., Mellempudi, N., Oberman, S., Shoeybi, M., Siu, M., and Wu, H. Fp8 formats for deep learning, 2022.
[32] Moon, F. C. *Chaotic and Fractal Dynamics: An Introduction for Applied Scientists and Engineers*. John Wiley and Sons, Inc., 1992.
[33] Omni OpenMP Compiler Group, University of Versailles Saint Quentin en Yvlines. Nas parallel benchmarks 3.0—unofficial openmp c version. https://github.com/benchmark-subsetting/NPB3.0-omp-C, 2014.
[34] Popek, G., and Goldberg, R. Formal requirements for virtualizable third generation architectures. *Communications of the ACM* (July 1974), 413–421.
[35] Rubio-González, C., Nguyen, C., Nguyen, H. D., Demmel, J., Kahan, W., Sen, K., Bailey, D. H., Iancu, C., and Hough, D. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing)* (2013).
[36] Walker, J. Fbench: Floating point benchmarks. https://www.fourmilab.ch/fbench/, September 2021.
[37] Walker, J. Ffbench: Fast fourier transform benchmark. https://www.fourmilab.ch/fbench/ffbench.html, January 2025.
[38] Wanninger, N., Dhiantravan, N., and Dinda, P. Virtualization so light, it floats!: Accelerating floating point virtualization. In *Proceedings of the 34th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC)* (July 2025).
[39] Wingo, A. Value representation in javascript implementations. http://wingolog.org/archives/2011/05/18/value-representation-in-javascript-implementations, May 2011.
[40] Zhu, F., Gong, R., Yu, F., Liu, X., Wang, Y., Li, Z., Yang, X., and Yan, J. Towards unified int8 training for convolutional neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2020)* (June 2020).

A Artifact Description

We contribute the following research products:

- Linux 5.15.0 kernel module which implements the lazy FPU abduction concept and exposes it to userspace programs.
- Modifications to the open source FPVM floating point virtualization framework to interface with the aforementioned kernel module, and to implement the low-precision "teeny" alternative arithmetic system.

Both of these contributions are described in detail throughout the main paper, and An unmodified version of FPVM is publicly available at https://github.com/PrescienceLab/fpvm.

B Artifact Evaluation

B.1 Hardware/OS Environment

As described in Section 6, all data was gathered using a Dell R6515 containing a single AMD EPYC 7443P 24-Core processor, a single NUMA zone, 64 GB of RAM, and support for the SSE4.2 and AVX2 floating point instruction sets. The operating system used was Ubuntu 22.0.4.5 LTS with the 5.15.0 Linux kernel, and our custom kernel module added via `insmod`. It may be possible to use a hardware environment which is merely similar to the one described above as FPVM is relatively portable across x64 machines (though performance results may vary). However because of the large number of kernel internal API's which are hooked/probed by the lazy FPU abduction kernel module, it is highly unlikely that the kernel or operating system versions could be substituted without issue.

B.2 Obtaining the Artifact

A tarball of the artifact directory, including the source code of our kernel module, modified version of FPVM, and a set of helper scripts can be found at https://doi.org/10.5281/zenodo.18501832.

B.3 FPVM Dependencies

FPVM and its build-system have a number of dependencies, the following commands will install them on Ubuntu 22.0.4.5:

```
sudo apt install build-essential libcapstone-dev \
                 libmpfr-dev libhdf5-dev python3 \
                 python3-pip git
pip3 install --user -r ./fpvm/requirements.txt
```

B.4 Building/Installing the Kernel Module

Before any of the benchmarks can be run, the lazy FPU abduction kernel module (dubbed "fptrapall" in the artifact) will need to be built and installed. To make this simple we have provided two scripts, `./fptrapall_build.sh` and `./fptrapall_setup.sh` to do both of these tasks. The setup script will need to be passed the name of a user group which can be given access to the interface between the kernel module and userspace, and any user which would like to run FPVM with "teeny" values will need to be a member of this group.

B.5 Capturing Performance Results

Once the kernel module is installed, running

```
make benchmarks
```

will build FPVM and all[9] of the benchmarks seen in the paper. It will then run these benchmarks under FPVM with 11-bit exponents and 38-bit mantissas. All of the data captured while doing so will be saved in a new `./data` directory.

Once all of the benchmarks have completed, the command

```
make plots
```

will generate all of the performance based plots seen in the paper and save them as PDF files in the `./data` directory.

B.6 Capturing Heatmap Results

The heatmaps in Figures 5 and 6 are generated by running each of the benchmarks under FPVM while varying the environment variables:

```
FPVM_TEENY_EXP_BITS and FPVM_TEENY_MANT_BITS
```

Because we generate these graphs for 3 benchmarks, this takes $3 \cdot 11 \cdot 38 = 1254$ runs, each of which has a timeout set at 5 minutes per run, though usually runs finish in less than a minute for these benchmarks (SP, FBench, and Lorenz). This means a full run can take a maximum of 104 hours. Though with the specific hardware environment used by the authors, all of the data was collected in less than 12 hours (left to run overnight).

With the lazy FPU abduction kernel module installed, all three figures can be generated by running

```
make heatmaps
```

This will result in many files being generated in the `data/BENCHMARK/` directory capturing the exact output of each benchmark run. These output files are then parsed by

```
scripts/BENCHMARK_make_heatmap_csv.sh
```

which is written specifically for the output format of each benchmark to determine the error of a given run. This generates a CSV file with all of the raw data for each heatmap and renders the heatmap to a PDF, both of which can be found in the `./data` directory.

[9] We do not automatically build Enzo due to this benchmarks rather complicated build-system/configuration. Instead we have provided a pre-built binary of Enzo for the x64 architecture. If this is undesirable, then the file "./enzo/enzo.exe" can be replaced with a version built from source separately.