FPVM: Towards a Floating Point Virtual Machine

Peter Dinda
Northwestern University

Nick Wanninger
Northwestern University

Jiacheng Ma
Northwestern University

Alex Bernat
Northwestern University

Charles Bernat
Northwestern University

Souradip Ghosh
Northwestern University

Christopher Kraemer
Northwestern University

Yehya Elmasry
Northwestern University

Abstract

Alternatives to IEEE floating point arithmetic have become all the rage. Some extract more representational power out of the available bits. Others offer the potential for lower or higher precision than is available in IEEE-compatible hardware. Even an "interface to the real numbers" has recently been proposed. Using such alternative arithmetic systems within an existing scientific or other significant codebase is a major challenge, however. We explore how to address this challenge through virtualizing the IEEE floating point hardware, specifically on x64. The goal of the floating point virtual machine (FPVM) is to allow an existing application *binary* to be seamlessly extended to support the desired alternative arithmetic system with overheads determined by that system and not the virtualization mechanisms. We describe the prospects, issues, and tradeoffs for four different approaches for building FPVM: trap-and-emulate, trap-and-patch, binary transformation, and IR transformation. We then describe the design and implementation of our current design, which combines static binary analysis/translation and trap-and-emulate execution. We evaluate our FPVM implementation on several benchmarks, virtualizing them to use posits and MPFR. Finally, we comment on kernel- and hardware-level innovations that could further reduce overheads for floating point virtualization.

CCS Concepts

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

Keywords

floating point arithmetic, virtualization, software development, IEEE 754

This project was supported by the United States National Science Foundation via grants CNS-1763743, CCF-2028851, and CCF-2119069.

HPDC '22, June 27-July 1, 2022, Minneapolis, MN, USA

ACM ISBN 978-1-4503-9199-3/22/06…$15.00
https://doi.org/10.1145/3502181.3531469

ACM Reference Format:
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 (HPDC '22), June 27-July 1, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3502181.3531469

1 Introduction

Virtually all applications in scientific and engineering domains, as well as applications built on machine learning techniques, make extensive use of IEEE 754 floating point arithmetic [32, 33] through its numerous implementations. Floating point has proven to be extremely effective at enabling high performance while providing behavior that is sensible to a knowledgeable developer.

Motivation: The preeminence of IEEE floating point hardware implementations is being challenged along three fronts. First, alternatives such as unums/posits [26, 37], BFloats[38], logarithmic arithmetic [3], and others [29, 43] potentially extract more useful representational power out of the same number of bits, or have range/precision tradeoffs that are more suitable for some modern workloads such as machine learning. The second front involves using these representations, as well as IEEE floating point arithmetic (for example in GNU MPFR [23] or libBF [7]), at arbitrary precisions, including much higher precision than the hardware directly implements. Finally, there are proposals to rethink floating point and related representations altogether in favor of an API to the real numbers [11]. Such an API would allow programmers to reason about their code using the rules of standard arithmetic and achieve reasonable performance in many cases. This approach (or higher precision) might also mitigate the effects of misunderstandings developers have about various aspects of IEEE floating point [18, 20].

Limitations of state-of-the-art approaches: Despite their benefits, using alternative arithmetic systems within an existing scientific or other significant codebase is a major challenge. A nightmare scenario is having to rewrite the application using a new API. A more pleasant scenario is when the programming language supports pluggable number representations, such as Fortran 90's `kind` parameter for type specification, or the recent VPFloat [35, 36] extension to C++. In this case, the programmer needs to modify much less source code, but they still must deal with cross-language issues (if even possible) and update and rebuild any libraries their codebase uses. Of course, these become daunting tasks for a large application. Additionally, any freshly rebuilt application may need

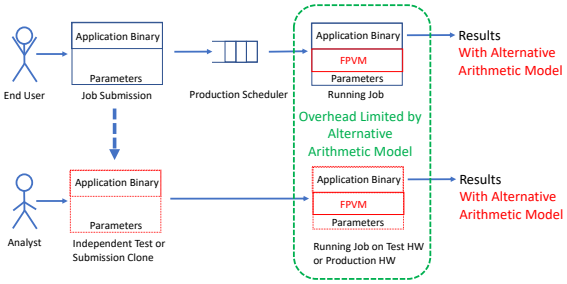

Figure 1: Desired FPVM model.

re-certification if there are critical dependencies to it. What if we want to use alternative arithmetic in an existing, trusted binary?[1]

There are also a large range of tools (e.g. [6, 8, 9, 14, 15, 22, 39, 42, 44, 48, 52–54]) whose goal is to improve the quality of the source code, generally by identifying sections that have high dependence on precision or on compiler/hardware optimization choices. These sections may have numerical stability problems that are due to algorithmic design and implementation issues, or where optimizations are buggy or change the semantics of the source code. Tools such as the ones listed often incorporate shadow arithmetic that is done using a different precision than the original code. Some of the tools operate on existing application binaries, avoiding the issues with source-level approaches to using alternative arithmetic described above. However, because the tools' goal is to point out problematic code to the programmer, they typically have substantial performance overheads. Execution under one of these tools quickly builds code coverage, making this less of a concern.

Key insights and contributions: We propose an alternative approach, namely virtualization of IEEE floating point hardware. Existing, unmodified application binaries could be run in a Floating Point Virtual Machine (FPVM) with the user choosing the desired alternative arithmetic system when the program is run. Each instruction would run directly on the underlying hardware at full speed provided the instruction's arithmetic does not create an imprecise result. When an imprecise result occurs, the instruction would instead be executed via the alternative arithmetic system.[2] Data would flow through the dynamically executing instructions of the original binary in precisely the way it does using standard execution, with the floating point values in the program either serving as actual IEEE numbers or as proxies for the numbers in the alternative arithmetic system.

Our use of the term "virtualization" is no accident. General-purpose virtual machines have little to no overhead compared to native execution. Our goal is for FPVM to have similar performance characteristics. In particular, we want the cost of virtualizing the hardware floating point unit to be low enough that it is dominated by the cost of the alternative arithmetic system. Reaching this cost objective would make it practical to substitute alternative arithmetic and/or arbitrary precision much as we might choose to use virtualization to effortlessly gain the use of a more powerful remote server. Figure 1 illustrates the desired model. The top path shows the use in production, while the lower path shows the use by an analyst. In both cases, we want to be able to operate on the specific binaries used for production science, and we want the overhead of using FPVM to be the overhead of using the alternative arithmetic system.

Experimental methodology and artifact availability: In this paper, we describe our progress toward building an effective and performant FPVM. We examine four different approaches for building an FPVM as well as their specific benefits and tradeoffs. These approaches are trap-and-emulate, trap-and-patch, static binary analysis and transformation, and intermediate representation (IR) transformation. We then describe the design and implementation of a FPVM that uses a hybrid approach that combines static binary analysis and transformation with dynamic trap-and-emulate execution. The hybrid FPVM was then validated on several common floating-point benchmarks and then evaluated on select scientific applications.

Interestingly, much like the general purpose x86 ISA prior to the availability of hardware virtualization support, the x64 floating point ISA(s) and hardware are *almost* virtualizable: Some instructions unfortunately do not trap under all the necessary conditions. As a consequence, a completely trap-and-emulate approach, in which there is no overhead unless an alternative arithmetic value is produced or consumed, is not possible. Our hybrid approach uses static analysis and transformation to find any such instruction where a floating point value could flow. These locations are then patched with software checks to detect NaNs. In this way, we can track our NaN-boxed values even in those instances where hardware currently cannot do so. Similar to general purpose virtualization, we believe that hardware changes to allow the floating point unit to be "fully virtualizable" are possible.

Our detailed contributions are as follows.

- We outline the concept of a floating point virtual machine (FPVM) that can add alternative arithmetic to existing programs, ideally at the level of existing, unmodified binaries, and can do so with low virtualization overhead.
- We describe how NaN-boxing can be used as the vehicle for tracking alternative arithmetic values using the program's original floating point values, as well as the limits in this approach.
- We describe the prospects and tradeoffs of four basic approaches building an FPVM: dynamic trap-and-emulate, dynamic trap-and-patch, static binary transformation, and compiler-based approaches, in particular static IR-level transformation.
- We describe the design and implementation of a hybrid FPVM for x64 that is based on trap-and-emulate, but uses static binary analysis and transformation to handle cases that x64 hardware cannot currently detect.
- We show that our hybrid FPVM can be combined with several alternative arithmetic systems: emulated x64 floating point, posits, and arbitrary precision floating point arithmetic as implemented in GNU MPFR.

[1] We note that changing the arithmetic system used by a certified application binary might well require it to be re-certified for the results to be trusted. However, being able to run below a certified binary would allow for experiments in which only one variable—the arithmetic system—is changed.

[2] Another choice would be to always use the alternative arithmetic system.

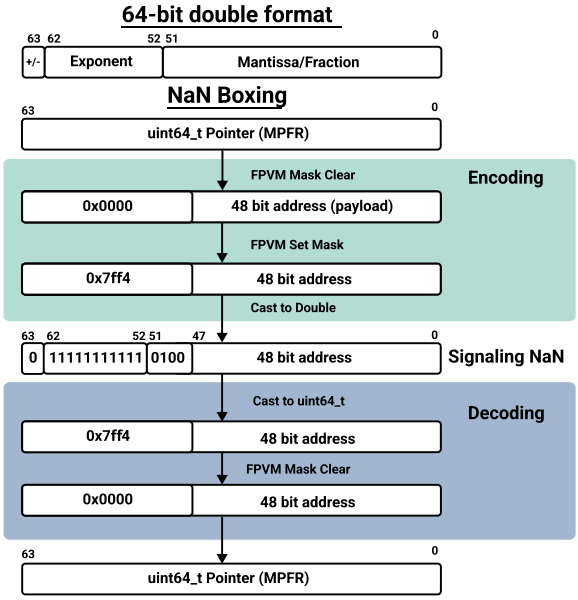

Figure 2: NaN-boxing of pointers using signaling NaNs.

- We validate our FPVM tool by running it with benchmarks and test codes from FBench, NAS, Mantevo, Enzo, a Lorenz system simulator, and a three-body problem simulation.
- We analyze the overhead of our tool on the same test codes, considering the cost of floating point virtualization and the overall effect on performance.
- We apply our FPVM tool to the test codes where higher precision is likely to change results due to modeling of chaotic dynamics, primarily Lorenz and three-body.
- We describe changes to the hardware and operating systems layers that would further reduce the overhead and avoid the use of static analysis and transformation.

Our FPVM prototype is not yet available, but we intend to make it publicly available via presciencelab.org.

Limitations of the proposed approach: As noted above, there are two current limitations to a trap-and-emulate approach to floating point virtualization, such as our prototype. The first is the high hardware and kernel costs involved in delivering a trap on x64. The second is that the fact that x64 floating point hardware only partially meets the requirements for classical virtualization, necessitating a hybrid approach. Section 3 compares and constrasts our approach with others, while Section 6 describes hardware and kernel changes that would reduce the overhead.

2 NaN-boxing

Conceptually, in an FPVM system, both original floating point values and values in the alternative arithmetic system co-exist, with promotion/demotion between them occurring on an as needed process. Whenever possible, an instruction is executed directly by the hardware using its original unpromoted input operands and producing unpromoted output operands. If the output cannot be computed exactly[3], the instruction is emulated using the alternative arithmetic system and the output operands are promoted. When an instruction uses a promoted input operand, the instruction is similarly emulated.

All of our approaches to floating point virtualization thus share the common need of being able to track the flow of a promoted value from instruction to instruction in precisely the same way that the unpromoted value would have flowed. Additionally, we would like to readily and cheaply identify when a promoted value is used as an input operand of an instruction. To achieve these requirements we use the technique of NaN-boxing. With NaN-boxing, a given floating point value in the original program, whether in memory or in a register, can represent one of two things:

- An *unshadowed value*, which is an original floating point value that is not a NaN.
- A *shadowed value*, which encodes a pointer or key that is used to find the corresponding value (the *shadow value*) within the alternative arithmetic system that is being used. The shadowed value is encoded as signaling NaN.

As we describe later, the x64 hardware, and others, can be configured to trap when encountering a signaling NaN, and hence can detect when shadowed values are used in our scheme. Note also that NaN values (and hence shadowed values) flow through the program directly in place of the IEEE floating point values.

NaN-boxing [13, 58] is technique that is used in some high-level language implementations. The observation is that the IEEE standard floating point values have numerous possible representations of a NaN. For a 64 bit floating point value, 52 bits are used for the mantissa, and 11 for the exponent. A NaN is encoded by setting the exponent bits to all be high, and allowing for at least one bit of the mantissa to be nonzero. It is common to differentiate signaling NaNs (those that can raise an exception when generated or used) and quiet NaNs (those that do not do so) through the use of the high-order bit of the mantissa (if it is set, we have a signaling NaN). Suppose we require only signaling NaNs. As can be readily seen, there are 2^{51} possible encodings of a signaling NaN.[4]

NaN-boxing makes use of this flexibility to encode up to 51 bits of extra information into a signaling NaN. In some JavaScript implementations, for example, all scalar values are encoded at their base as 64 bit floating point values. Integers and pointers (e.g., strings) are encoded by creating a NaN and embedding the value as the 51 extra bits that are available. When handed a scalar value, the JavaScript interpreter can readily check its encoding to determine if it is an actual floating point value, a NaN, or some other type by simply checking the exponent and two bits of the mantissa.

We use NaN-boxing to encode pointers (shadowed values) to alternative arithmetic values (shadow values) into the existing floating point values that are used by x64 floating point instructions. This encoding is such that the "NaN" will be interpreted by the

[3]For example, if the original instruction causes an overflow, underflow, or rounding event.

[4]It is important to note that while the size of the x64 virtual address space is technically 2^{64}, given canonical addressing, only a fraction of this can actually be used. In Linux, the user portion of the address space (< 48 bits, currently) readily fits into 51 bits at this time. On a different platform, or if this ever ceases to be the case on x64, the shadow value allocator could be implemented to use a chunk of address space that can be addressed with 51 bits, or the 51 bits could simply be used as a key to a hash lookup scheme instead of directly as a pointer.

hardware as a signaling NaN, resulting in an exception whenever such a value is used by an instruction. This exception is ultimately handled by FPVM, which decodes the "NaN" into the embedded pointer to find the corresponding shadow value. If a new shadow value is generated, the pointer to it is encoded into a signaling NaN. Figure 2 illustrates our specific decoding and encoding processes.

Limitation: NaN-space ownership: In our scheme, all signaling NaNs are "owned" by FPVM. If the program itself is using signaling NaNs, it will still operate, but will never "see" a signaling NaN.

Limitation: universal NaNs: Regardless of the arithmetic system involved, some computations simply do not result in real numbers. For example, $\frac{0}{0}$ is not a real number and thus will be a NaN in any alternative arithmetic system. How such a universal NaN can be made visible to the original program is unclear. One possible approach is to treat any signaling NaN without a corresponding shadowed value as a universal ("true") NaN.

Limitation: float problem: Our scheme is designed for 64 bit (and larger) IEEE floating point numbers. Of course, NaN-boxing can also be applied to smaller numbers, such as `floats`. However, given that their mantissas are much smaller (23 bits for `float`) the number of distinct pointers or keys that can be encoded is likely to be insufficient.

Limitation: externals: Floating point virtualization should ideally be independent of the structure of code within a virtualized process. However, this is quite challenging for some of our approaches. If there is a boundary in the process between code that is under FPVM control and code that is not, the boundaries need conversions from shadowed values to regular floating point values. This is a particular issue for the static patching and compiler-based approaches.

Limitation: printing problem: Standard output facilities, such as `printf` are specifically designed to convert an IEEE floating point representation to a human-readable string. Of course, for shadowed values, these tools would simply see signaling NaNs. To handle this, we must hijack such output functions to call back to our output code, for example, to promote "`%lf`". Section 4.2 describes our technique for this.

Limitation: serialization problem: Code that writes floating point values to storage or to a network connection will instead be writing shadowed values (i.e., the NaN-box encoded signaling NaNs. While in principle FPVM could transport the shadow values alongside the shadowed values and reconstruct the result at the destination[5] or in the file, this is in effect the deep copy problem from RPC. Another approach that could be taken is to do conversion back to IEEE floating point values at the point of serialization, but this would entail losing all the promoted values.

3 Approaches to floating point virtualization

We considered four approaches to building an FPVM. These points in the design space have tradeoffs with respect to code coverage, ease of use, static and dynamic overheads, software engineering focus, and other aspects that are summarized in Figure 3. The hybrid FPVM described in Section 4 adopts the trap-and-emulate approach, but uses simple static analysis-and-transformation to work around the fact that the x64 floating point is not (yet) fully virtualizable.

3.1 Trap-and-emulate

The trap-and-emulate approach takes its inspiration from how a classic virtual machine or hypervisor operates [25, 56], although it can (and should ideally) operate entirely at the same privilege as the application, since protection is not a concern. Whether it is user-level or kernel-level software, it is also unconcerned with interrupts. It is driven entirely by exceptions caused by floating point instructions and related instructions in the ISA. In effect, this FPVM approach adds the floating point virtualization capability to an existing process abstraction. This also makes it readily applicable to any code that a process executes. This is a major plus from a requirements perspective—if the user has something that runs, we can virtualize its floating point operations.

The formal requirements for hardware to support virtualization are well-known [49]. For the trap-and-emulate approach to floating point virtualization when Section 2's NaN-boxing approach is used, the general requirements simplify: the hardware requirement is that a floating point instruction that consumes a NaN, or produces a rounding, denormalization, underflow, or overflow event must trap so that FPVM is invoked. This requirement is partially met already by the x64 hardware, and in our hybrid approach (Section 4) we patch instructions where this is not true, similarly to how VMware handled the partially virtualizable x86 general purpose ISA prior to 2005 [17].

For a dynamic floating point instruction that does not trap (and is not patched), there is no overhead—it simply executes as normal. If the instruction does trap, FPVM is invoked to decode the instruction and emulate it using the alternative arithmetic system.

Shadowing and garbage collection: Unlike instruction emulation in a general-purpose VMM, a trap-and-emulate FPVM must maintain essentially arbitrary shadow values (numbers represented in the alternative arithmetic system)—these are *not* constrained to privileged register state as in a VMM. Moreover, temporary values are included. Conceptually, each time an instruction is emulated, a new shadow value is potentially produced. While the shadowed value (the NaN-boxed pointer to the shadow value) is implicitly "garbage collected" by normal function execution,[6] these events are *invisible* to a trap-and-emulate FPVM. As a consequence, shadow values accumulate in FPVM and must be explicitly garbage collected by it.

3.2 Trap-and-patch

The trap-and-patch approach extends and builds upon the trap-and-emulate approach, and can offer the same ideal interface to the user. Trap-and-patch uses the same hardware trap mechanism as trap-and-patch, but instead of decoding and emulating the faulting instruction, it instead replaces the instruction with a new instruction sequence (the *patch*), and also dynamically generates a *custom handler* for the instruction being replaced. While these two elements can be combined, the patch is typically size-constrained so

[5] By interposing on MPI calls, for example.

[6] For example, on a return instruction, variables on the stack frame values, are garbage collected. As another example, when an instruction is executed twice with the same destination operand, the operand is overwritten and the old value is discarded.

Aspects	Approaches			
	Trap-and-emulate	Trap-and-patch	Static analysis-and-transform	Compiler-based-transform
Code supported	**all (any process)**	**all (any process)**	complete binaries that are statically available	complete IR / source code that is statically available
User requirements	**none**	**none**	must provide *all* binary code (libraries) before use	must provide *all* IR code or source code before use
HW requirements	Fully virtualizable FP (or selective patch)	Fully virtualizable FP (or selective patch)	**none**	**none**
Static costs (compilation)	**none**	**none**	huge	large
Run-time overhead when alternative arithmetic not involved	**none**	low	low	low (<binary approaches)
Run-time overhead when alternative arithmetic involved	high (but OS+HW dependent, see §6)	low	low	**low** (<binary approaches)
Hardware-independent	no	no	no	**yes**
Major SE focus	RT/OS	RT/OS/JIT	Binary analysis/transform tool	Compiler

Figure 3: Comparison of the approaches. Boldface indicates most desirable properties.

that it can be easily inlined, while the custom handler is a separate function.

The next time the instruction site is encountered, no hardware exception can occur. Instead, the patch invokes the custom handler. The handler does a *precondition check* that determines if any of the input operands to the replaced original instruction are a NaN. If not, then the original instruction, embedded in the handler, is executed, knowing it will not fault. The handler then does a *postcondition check* to see if the result was rounded, underflowed, overflowed, etc. If not, then execution returns to the normal flow subsequent to the patched instruction. If either check fails, the custom handler includes instruction-specific code that invokes FPVM internals to emulate the original instruction. These checks are effectively the same checks as the hardware performs in trap-and-emulate.

The trap-and-patch approach is substantially more complex than trap-and-emulate because it is doing architecture-specific binary code generation for both the patch and the custom handler. Additionally, it must be able to apply the patch regardless of the size of the original instruction. However, if the original instruction were to frequently see or produce shadowed values, trap-and-patch can operate with much less overhead than trap-and-emulate. The primary reason why is that the delivery cost of a floating point trap generated by the hardware is currently substantially higher than the cost of executing the patch, detecting a precondition/postcondition check failure, and calling into the handler. On the other hand, if the original instruction rarely sees or produces shadowed values, trap-and-emulate will be cheaper because software checks always have an overhead, unlike the hardware checks, which only have overhead only when they fire.

On a typical RISC architecture, patch generation is simplified due to the simple instruction sizing, and the patch can essentially be a single instruction. On x64, however, it is nontrivial because instructions may be short. In particularly, if the instruction is five bytes or longer, it can be replaced with a relative jump or call to the custom handler. If it is shorter than five bytes, the patch needs to span multiple instructions and may straddle the end of a basic block. These are not showstoppers, and there are tools that accomplish such patches (e.g., DyninstAPI [30]), but engineering complexity now expands to include those tools.

To consider the prospects of trap-and-patch, we developed a simple proof-of-concept that would create a patch and handler for a simple x64 SSE addition operating on a pair of registers. Our goal here was to measure the runtime overheads of a patch+handler operating when the conditions are met versus not. These numbers are included in Section 5.

Note that the software engineering effort of trap-and-patch is focused on the JIT code generator needed to produce the patch and handler. Operating at this level—writing the code to generate the code that will be used at run-time—is substantially more error prone and difficult to debug. However, given a working trap-and-emulate system, the trap-and-patch approach could be developed incrementally.

3.3 Static binary analysis and transformation

In this approach, instead of the trap-and-patch model of dynamically patching the running process with invocations of FPVM, we instead statically transform the application binary so that all instructions that could possibly employ a shadowed value are patched. This instrumented binary is then linked with FPVM and used in place of the original. At runtime, no hardware checks are used at all. Because the transform is static, it is simpler than dynamic patching, and there is a wider range of tools, such as e9patch [21], which we employ later, that can be leveraged. A deficiency is that all instruction sites now include the overhead of the software condition checks regardless of whether they ever trigger. Another deficiency is that the static analysis and transformation process can be very time consuming, and likely needs to be repeated whenever the binary changes.

It is important to point out that with this approach, it is the binary, as opposed to the process, that is patched. As a consequence, it is necessary for all code that the program could ever use to be available to the static transform process. This a tall order for the user because most binaries use dynamic linking aggressively. Furthermore, larger scientific applications are likely to use more dynamically linked libraries. There is also no guarantee that all dynamic libraries will be visible as imports in the application binary—a shared library can always be explicitly linked into the program at runtime using a name available only at runtime.[7] Finally, even if the entire binary is statically linked, some libraries, such as `libm.a` and the GNU standard C library still do architecture-specific code selection via an internal patching scheme.

[7]The class loader interface of Java is one example.

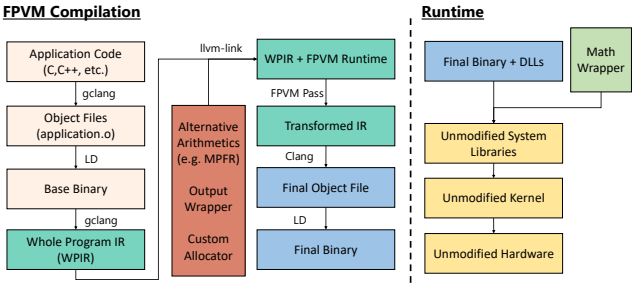

Figure 4: Compiler-based approach.

3.4 Compiler-based

The compiler-based approach requires that intermediate representation (IR) code (including for all libraries) is available. If only source code is available, it is first compiled into IR. Note that this increases the challenge for the user well beyond that of the static analysis and transformation approach because they must now collect IR code or source code. That said, on some systems (e.g. Apple), code is starting to be distributed in an IR representation to facilitiate optimizing it for the specific machine at install time. This would reduce the challenge.

The IR represents the program in an abstract, architecture independent manner that is suitable for analysis, transformation, optimization, and the generation of architecture-specific object code. We specifically use LLVM IR [41], which is a static single assignment form (SSA) using a relatively small abstraction instruction set. In contrast to something like the x64's several(!) floating point ISAs, their hundreds of instructions, and handful of encodings, there are only a tiny number (13) of LLVM IR instructions that we are concerned with. There exist widely used third-party tools that complement LLVM, such as WLLVM [51] and gclang [31], which allow us to extract whole-program IR of the entire program with extremely small modifications to the build process. Using the whole-program IR and building from it greatly simplifies the transformation step (as there is only one IR module to transform) and produces more opportunities for successful interprocedural analysis and optimization.

The compiler-based approach transforms the code at IR level to introduce the equivalent of the patches and custom handlers introduced by the static analysis and transformation approach. Because the LLVM IR is so much simpler than any ISA, this transformation involves far less engineering effort. Additionally, the transformed code can be subjected to another round of optimization which may be able merge the patches and handlers with application code, thus reducing their overheads compared to binary patching. The runtime system for the approach is essentially the same as for the other approaches. Figure 4 illustrates the compiler-based approach as we have implemented it in an initial prototype. The compilation aspects of the VPfloat system [35, 36] are similar, except they make the "pluggable float" type explicit in the language (C++). The programmer also needs to modify the source to use it. In a compiler-based FPVM, the compiler and a complementary runtime would manage shadowing of objects in the alternative arithmetic system and their allocation/deallocation via static analysis and transformation–also similar to the VPFloat system's backend code generators.

While rebuilding the entire codebase is a substantial disadvantage for our usage scenario compared to the other approaches, the compiler-based approach has two substantial advantages as well. First, the compiler's code generator can easily target a different ISA, which means targeting a different processor (e.g., ARM, RISC-V, possibly GPU) does not require a new engineering effort. The other approaches require rebuilding much of the system to support a different processor. Second, the transformation can take into account the connection between shadowed values and shadow values more cleverly. In particular, it knows exactly when a program temporary is garbage collected, and thus can easily add a callback to the runtime to also free the shadow value. This can substantially simplify garbage collection within FPVM, lower the overhead of garbage collection, and reduce memory overheads. Finally, at the compiler-level there are additional opportunities to merge and/or reuse shadow values through the use of liveness analysis. Such optimizations would reduce overheads.

4 Hybrid FPVM for x64

Our prototype hybrid FPVM system is designed to run at user-level on top of an unmodified x64 platform running an unmodified Linux kernel. It runs underneath an existing application binary. The core of the implementation is a trap-and-emulate engine akin to a classical VMM that allows an alternative arithmetic system to be used. Before the binary can be used, however, it must be run through a static analysis and patching process to catch corner-cases that trap-and-emulate will fail to catch on its own. An abstraction layer allows alternative arithmetic systems to be ported for use with FPVM.

4.1 Trap-and-emulate engine

The core trap-and-emulate functionality of our FPVM implementation leverages the ideas behind our FPSpy analysis tool [19]. As with FPSpy, FPVM is implemented as an `LD_PRELOAD` library, loaded by the dynamic linker at program launch time before all other libraries. This allows FPVM to insert itself in front of any other part of the runtime, effectively acting as a shim between the application binary and its libraries. Like FPSpy, FPVM also installs itself as the handler of `SIGFPE` signals, which result when the floating point hardware detects exceptional conditions. FPVM manages the floating point hardware control state such that these conditions are configured to be detected and to result in hardware faults.

FPSpy responds to a hardware fault and the resulting `SIGFPE` by *recording* the execution of the faulting instruction, and then allowing it to be executed as normal. In contrast, FPVM responds to the same situation by *emulating* the faulting instruction using the alternative arithmetic system, and storing pointers to the shadow results in the process's memory and/or registers using the NaN-boxing technique of Section 2. The runtime component the hybrid FPVM is broken down into four main components: trapping, decoding, emulating, and garbage collecting. An overview of the architecture is shown in Figure 5.

Trapping: The x64 floating point hardware includes a control and status register, `%mxcsr`, that maintains a set of condition flags

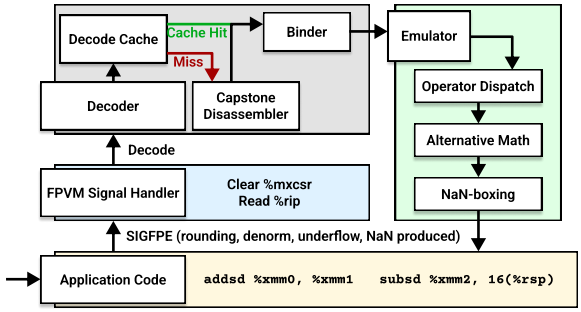

Figure 5: Hybrid FPVM prototype's trap-and-emulate architecture

defined by the IEEE floating point specification. Unlike integer condition codes, these flags are *sticky*, meaning they must be manually cleared by software. FPVM manages these flags so that they start at zero for each instruction. The flags record a small set of events (result was rounded, result overflowed, result underflowed, result was denormalized, and NaN was produced or consumed). `%mxcsr` also contains parallel exception mask flags for each of these condition flags, that, when clear, cause a precise exception (a fault) to be raised to the kernel if the corresponding condition code flips to one (ie., when the event occurs) during the execution of an instruction.

The kernel translates this exception into a `SIGFPE` signal to the process. The signal is delivered to the handler FPVM previously installed. FPVM inspects and records the `%mxcsr` register to determine the reason for the signal, and then clears the condition codes within in preparation for the next instruction. The signal handler then reads the instruction pointer from the kernel-provided signal trap frame to determine the location of the faulting instruction.

Decoding and decode cache: Once the address of the faulting instruction is known, it is then fed into the decoding subsystem. This code keeps a cache of decoded instructions–a map from address to `struct instruction`–that is quickly queried to avoid decoding the same instruction multiple times. This decode cache is critical to lowering latencies, as is discussed in Section 5.3.

If there is a cache miss, the decoder invokes the Capstone disassembler [2] to decipher the x64 instruction. It then simplifies the decoding for use specifically in floating point emulation. The resulting `struct instruction` contains both a high-level, Capstone-independent representation, and the low-level Capstone-dependent representation. The rest of the system uses the Capstone-independent representation to allow for future plugability of decoders. The hundreds of different x64 floating point instructions flatten down to about 40 operation types. The Capstone-independent representation is also designed to minimize architecture-specificity with the eventual goal of supporting architectures other than x64. That said, due to the challenges of representing the side-effecting nature of some x64 instructions, this is a work in progress. Once decoded, the instruction is then placed in the decode cache.

Binding: The `struct instruction`, whether it came from the decode cache or is a new instruction, next has its operands "bound" to memory locations. A bound instruction is an abstract normalized representation, containing direct pointers to the sources and destinations of the instruction, the size of the values being operated on, a simplified op-code which is later used for emulation, and any special details (like side effects). For example, the instruction `addsd %xmm0, 0(%rsp)` and `addsd %xmm0, %xmm1` are bound into the same `FPVM_OP_ADD` operation, where the former's source value points to the stack, and the latter's to the register file saved by the signal handler.

By explicitly binding each instruction, the construction of the emulator is vastly simplified. The emulator need not handle accesses to memory or registers differently, it only needs only read/write through a `void*`. The details of registers, immediates, and the complex x64 memory operand address computations, are hidden.

Emulation: Once the instruction has been decoded and bound, the emulator is invoked. Recall that the instruction's Capstone-independent representation marks it as having one of about 40 operation types. The implementation for each operation type is given simply by a function pointer stored in a map, `op_map`, which indexed by the operation type. These functions constitute the core of the interface to the alternative arithmetic system, which is discussed in more detail in Section 4.3.

By abstracting these operations, only a single scalar function needs to be implemented to handle all forms of an instruction like "add". For example, to handle a vector instruction, the emulator simply calls the function multiple times with different source and destination pointers.

These functions all fundamentally operate in the same way. They first attempt to unbox the values stored in the source operands. If the source registers are not NaN-boxed values (shadowed values), they are promoted from their double representation to the alternative arithmetic system's representation. The alternative arithmetic system's implementation of the instruction is then carried out on these promoted values. The resulting shadow value is then stored in a newly allocated cell which is NaN-boxed into the pointer (creating a shadowed value) provided by the decoder. Because FPVM must maintain the illusion that the numbers that the application is operating on are values, not pointers, the NaN-boxed data must remain immutable. For example, if a NaN-boxed reference is written to every location in an array, mutating the value will indirectly modify every value in said array. This unfortunately leads to significant memory pressure, as every instruction allocates a new cell.

Garbage collection: In order to tame the memory demands of FPVM, a garbage collector must be utilized to reclaim references that are no longer stored in NaN-boxes (i.e., to delete any shadow value that no longer has any shadowed value in the program to point to it). This garbage collection problem is not the general garbage collection problem, however, since there is no general pointer graph. Instead, the pointer graph is bipartite, between potential shadowed values in the program, and shadow values in FPVM. Conceptually, any memory location in the program could contain a pointer (a shadowed value), but no memory location (a shadow value) in FPVM can point back into the program. An additional simplification is that any reference held by a NaN-box is guaranteed not to point into the middle of an object.

As a consequence, a relatively naïve conservative mark-and-sweep collector is used. All allocations are stored in a simple data structure along side a "marked" bit. Every epoch (typically 1s), the garbage collector scans all *writable* program memory for data that appears to be a NaN-box. It then decodes it, and sets the mark bit if it is located in the data structure. It then sweeps through the set of

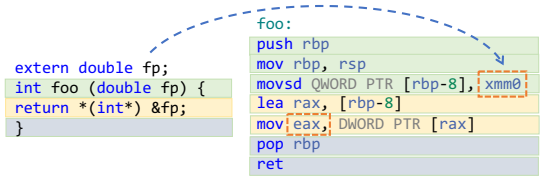

Figure 6: Double to Int conversion through pointers.

```
typedef struct A{
    int i;
    double d;
} A;
extern double fp;
int foo(){
A* ptr = malloc(sizeof(A));
ptr->d = fp;
ptr->i = 0;
return *(int*) &ptr->d;
}
```

```
call malloc
mov QWORD PTR [rbp-8], rax
movsd xmm0, QWORD PTR fp[rip]
mov rax, QWORD PTR [rbp-8]
movsd QWORD PTR [rax+8], xmm0
mov rax, QWORD PTR [rbp-8]
mov DWORD PTR [rax], 0
mov rax, QWORD PTR [rbp-8]
add rax, 8
mov eax, DWORD PTR [rax]
leave
ret
```

Figure 7: Double to Int conversion through pointers with struct.

all allocated values and frees their backing storage (shadow values) if they are not marked.

While an off-the-shelf collector such as Boehm [10] could be used instead, it would require significant patching to support NaN-boxing. It also would not take advantage of the simplifying properties of the specific garbage collection problem that arises in FPVM.

4.2 Static binary analysis and transformation

The correctness of FPVM's trap-and-emulate model requires that that all instructions involving NaN-boxed values are captured by our system. Unfortunately, some x64 instructions can operate on NaN-boxed values without triggering a hardware fault and thus FPVM. For example, developers have the ability to cast memory references to floating point values to integers to operate on their bits directly, as is done in much of the implementation of the GNU standard math library or in `printf`. Perhaps less obviously, modern compilers will often optimize common operations by operating on the bits of a floating point register directly, for example by flipping the sign bit using the `xorpd` instruction (vector xor operation). With current hardware, none of these instructions will trap when operating on NaN-boxed values, possibly leading to a sea of undefined behavior as the application begins to blindly operate on NaN values introduced by FPVM.

We illustrate an example of this behavior in Figure 6. Here a double-precision float's bits are reinterpreted through pointer casting. The argument, `fp`, is stored in `%xmm0` per the calling convention, which is then stored into a memory location on the stack. The content of this location is then loaded into the `%eax` register. If the value stored in `%xmm0` register were a NaN-boxed (shadowed) value, `%eax` now partially contains said NaN-box, possibly resulting in unexpected results if the application relies on or operates on the bit-pattern of `%eax`.

Whether or not the value stored in `%xmm0` was a NaN-boxed (shadowed) value, any bitwise operations on `%eax` will not fault, FPVM will thus be unaware of them, and this will likely lead to invalid results, or memory faults caused by NaN-box corruption.

Figure 7 is another variation of this issue, showing that the bit-level access can be complicated by indirection and structs/unions.

Correctly identifying instructions that might lead to this behavior is instrumental to maintaining correctness in FPVM. Given an oracle where all such instructions were known ahead of time, FPVM could defensively *demote* NaN-boxed values to their lower precision representations (IEEE doubles) in order to maintain correctness and transparency. Demotion not only means to switch back to the IEEE double representation, but also to store that actual value in place of the NaN-boxed shadow value. One solution is to demote NaN-boxed values *every* time they are stored to memory. This solution unfortunately obviates the goal of using the alternative arithmetic system, but guarantees correctness. What we really want to do is minimize the number of demotions needed to maintain correctness. In order to maintain program correctness while avoiding demotions, code analysis must be utilized to identify instructions that can produce problematic behavior.

Our approach to solve this problem is to statically analyze applications prior to running. Our static analysis is designed to identify vulnerabilities in programs when running with FPVM by tracking how data flows through all instructions and control flow. The analysis leverages Value Set Analysis (VSA) [5]. The analysis categorizes instructions into two categories: *sources* and *sinks*. In FPVM, a *source* is any instruction that stores a floating point value to memory, and a *sink* is any instruction that later loads from any memory location that was previously been written to by a source. Instructions that operate only on registers are not considered, as double→int reinterpretation in a register is not generated by either GCC or Clang/LLVM.

The input to our static analysis algorithm is an unmodified application binary, and the output is the set of sinks (instructions) at which FPVM must demote. Modern VSA often leverages symbolic execution to determine the possible values of any registers or memory at every instruction. By building on the information given by VSA, we can more accurately identify sink instructions, patching in demotions only as needed.

Unfortunately, as with most static analysis, such as alias analysis, VSA is not generally solvable [40, 50], and will not always give precise results. Thus, if VSA returns a conservative result, FPVM follows suit and assumes there exists a NaN-boxed double that may need demotion. For example, if an application calls into an external library external library that is not analyzed, the worst case must be assumed, and demotion is done at the call site.

Since the scientific applications FPVM targets are often huge, containing more than a million instructions (excluding external libraries), we tweak VSA under consideration of both running time and space consumption. First, our VSA treats each instruction as a basic block and associates a persistent state with each instruction. FPVM's VSA builds a preliminary Control Flow Graph (CFG) and then starts from the first instruction at the entry point and analyzes the program sequentially. Through symbolic execution, the set of possible values for each register/memory location is maintained.

After running VSA on the existing, unmodified application binary, we obtain states, including register states and memory states, before and after each instruction that was analyzed. Building on

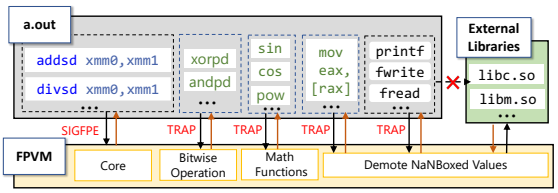

Figure 8: Overall model after static binary analysis and patching.

this, we identify sources and sinks defined above. Once sink instructions are identified, they are patched to explicitly trap into FPVM to demote the NaN-boxed value if it is discovered at run-time to truly be NaN-boxed, and then re-execute the instruction. For calls into external libraries, NaN-boxed values passed as arguments can be problematic, for example, when printing out floating-point values. Hence, we demote NaN-boxed floating point registers at the call site.

When running the patched binary, the dynamic interaction between the application, external libraries, and FPVM is as shown in Figure 8. What is different compared to Figure 5 is that FPVM is also invoked by the intentional traps introduced by static analysis for the reasons given above. If our analysis performs well, then these intentional traps which involve a overhead whether they are triggered or not, will be rare compared to the hardware-detected events, which only have an overhead when they occur.

FPVM's static analysis was built on top of angr [55], a tool that disassembles instructions, lifts into intermediate-representation, and carries out symbolic execution. Angr provides various static analysis techniques on top of its abstract register/memory representation. After VSA via angr converges on a result, FPVM's static analysis produces a list of sink instructions which must be patched to include traps to FPVM based on VSA results. We call those traps correctness traps. We use e9patch [21] to patch those instruction to explicitly trap to FPVM, where NaN-boxed values are demoted back to IEEE floats and the instruction is re-executed by using the x64's trap mode to do single instruction stepping.

4.3 Alternative arithmetic interface

FPVM includes an interface for alternative arithmetic systems to be plugged in. This parallels the abstract interface of the decoder, and consists of a small number (currently 37) scalar functions (the emulator handles vectors) that must be provided. 23 of these consist of arithmetic operations like add, multiply, multiply-add, sin, cosine, and square root, etc, 10 are conversion operations, and 4 are comparisons. Conversions and comparisons are the hairiest part of the interface as these require matching of the system to implicit input (e.g. rounding mode) and output (e.g. flags register) operands. FPVM also provides the alternative arithmetic system with memory management. We have thus far ported three alternatives to this interface.

Vanilla: This system implements the functions using regular IEEE 64 bit floating point operations. The primary purpose of Vanilla is to allow us to test the other elements of FPVM independently. If FPVM is working correctly, then Vanilla should produce the identical results to running without FPVM.

MPFR: This system interfaces to the GNU Multiple-Precision Floating-point Representation library [23]. MPFR is a widely used tool for arbitrary precision arithmetic. It essentially implements the IEEE floating point standard in software, but with dynamic runtime selectable precision. The fraction can be an arbitrary number of bits long, while the exponent is a 64 bit unsigned number. In our implementation, the precision used by FPVM is determined by a compile-time configurable parameter or environment variable, and we are also considering an adaptive precision version.

Posit: This system interfaces to the Universal Numbers Library [47] implementation of the posit standard [26, 37]. A posit number has four parts which include sign, regime, exponent and fraction. Among the four, exponent and fraction have variable length. The posit sizes/precisions available in the library can be chosen at compile-time.

5 Evaluation

We now describe the testing and initial performance evaluation of the hybrid FPVM prototype.

5.1 Testbed, benchmarks, and applications

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

Our test code consists of the FBench floating point benchmark [57], a version of the Lorenz system simulator that we developed, a three-body problem simulation, selections from the NAS 3.0 Application Benchmark Suite [4, 34, 46], miniAero, and an Enzo application. MiniAero is a Mantevo [16] miniapp (one of several used for evaluation of supercomputing environments by Sandia National Labs) that solves the compressible Navier-Stokes equation. miniAero is written in C++ and C and contains about 4400 lines of code. miniAero is dependent on kokkos for OpenMP and Pthreads. Enzo [12] is an astrophysics and hydrodynamics simulator. Enzo is written in C, Fortran, and Python and contains about 307,000 lines of code. Enzo depends on HDF5 for data storage, as well as an MPI library.

5.2 Validation

In order to validate the functionality of FPVM, we ran a selection of our codes with and without FPVM. When run under FPVM, we used the Vanilla math implementation outlined in Section 4.3. Recall that this simply interposes virtualization, but uses IEEE 64 bit floating point. In all of the cases, the results were identical, as expected, indicating that the core emulator operates correctly. As alternative math libraries such as MPFR are used, the results vary as outlined in Section 5.4.

5.3 Overheads

In the trap-and-emulate model, overhead is only incurred if the hardware detects an event (such as rounding, overflow, underflow, denorm, or use of NaN). When such an event is detected, the cost of executing the instruction expands to include FPVM. Figure 9 illustrates the average costs in this situation for our test codes, which range from 12,000-24,000 cycles. The figure also breaks these down

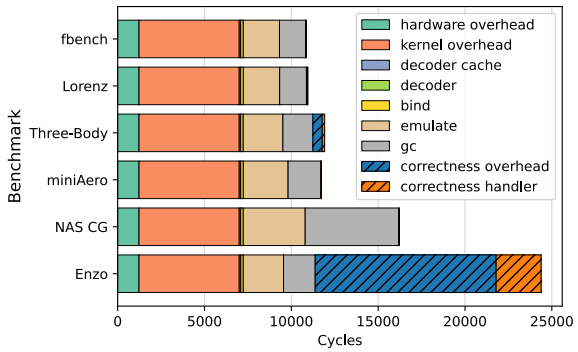

Figure 9: Average cost of virtualizing a floating point instruction, and its breakdown into constituant parts.

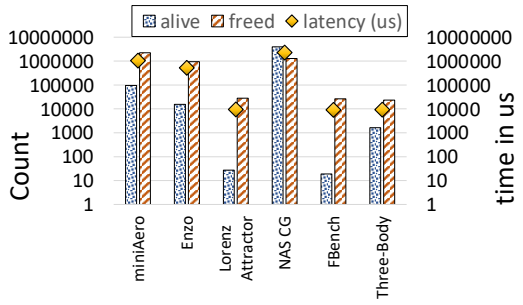

Figure 10: Garbage collector statistics and performance.

into their constituent components.[8] The emulation component includes MPFR computation with 200 bit precision.[9]

The correctness overhead and correctness handler components reflect the amortized dynamic cost of the trap instructions introduced via static analysis. These costs are virtually zero except for Enzo, where they are substantial. In Enzo, the traps occur in critical loops because the static analysis could not prove they were unneeded. The vast majority of the dynamic checks succeed however, meaning no special handling is needed. This gives hope that advances in our static analysis work could eliminate more of them. In contrast, miniaero's dynamic checks do not typically succeed, but they are not encountered in critical loops either. As a result, the correctness overheads are in the noise. The other codes are similar.

The "hardware overhead", "kernel overhead", and "correctness overhead" are the costs paid to dispatch into FPVM for a floating point exception or a correctness trap. As we describe in Section 6, these overheads are likely to become much smaller by kernel and hardware extensions. There is nothing intrinsic to them. We also note that the correctness overhead could be eliminated without kernel or hardware changes by having the static analysis patch in a direct call instruction to the FPVM entry point instead of a trap instruction or by inlining the dynamic check and invoking a trap only if it fails. This is only a matter of implementation effort.

FPVM operation generates considerable amounts of garbage due to the problem of temporaries noted above. Figure 10 measures the garbage collector behavior (> 95% of shadow values are collected on each garbage collection pass) and performance in more detail. Note, however, that this is not a dominant component of overhead—as Figure 9 makes clear, it is 2nd or 3rd order behind the kernel overhead, emulation overhead (similar to this), and the correctness overhead (on codes where this is significant). That said, there is plenty of room to enhance our garbage collector.

Recall that our goal with floating point virtualization is to have the overhead dominated by the alternative arithmetic system, and not the virtualization mechanisms. Of course, achieving this goal depends on the performance of the alternative arithmetic system.

[8]The decode component is the amortized cost over all faulting floating point instructions, and is very tiny because there are only a small number (typically 1000s) of these instructions, but they are executed millions to billions of times, thus the decode cache hit rate is nearly 100%.

[9]200 bit MPFR operations themselves take from 93 (add) to 2175 (divide) cycles.

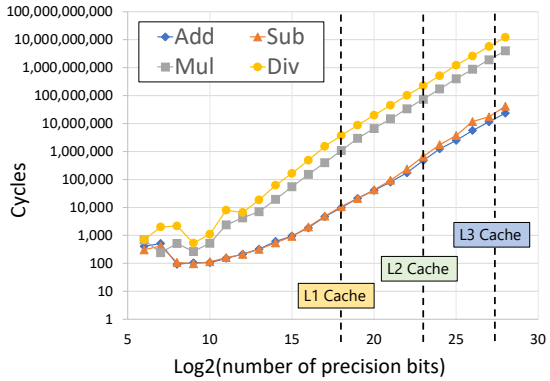

Figure 11: Performance of MPFR as a function of precision.

Figure 11 shows the measured performance of different MPFR operations, as a function of the precision (mantissa bit width). Assuming a crossover point at 12,000 cycles, for example, as with most of our codes, means that MPFR begins to dominate at 2^{13} bits (division) to 2^{18} bits (addition). This is prior to the optimizations we envision in Section 6 and above. With them[10], we would be left with about 4,000 cycles (dominated by emulation and garbage collection), where MPFR would begin to dominate at 2^8 bits (division) to 2^{16} bits (addition).

The 4,000 cycles we note above include emulation and garbage collection costs. It is possible that this cost could be further reduced through concurrent garbage collection techniques. Additionally, our emulation logic has not yet been optimized, and the measurements of Figure 11 include both the emulation cost and the cost of the Vanilla arithmetic system. We speculate that the ultimate overhead limit is substantially less than 4,000 cycles.

Figure 12 illustrates the wall-clock slowdown of each of our codes in the current implementation. Tests are done on three machines. R815 is as described in Section 5.1 and was used for the previous results. Notice that R815's slowdowns are substantially smaller than would be implied by Figure 9. This is because are many dynamic instructions other than floating point instructions or correctness traps. Furthermore, the floating point instructions only invoke

[10]In particular, user→user trap with fast delivery, extending floating point traps for all x64 instructions, and trap on NaN-load

Benchmarks	Specifics	Machine R815	7220	R730xd
FBench	n.a.	1,808x	720x	667x
Lorenz Attractor	n.a.	268x	116x	243x
Three-Body	n.a.	789x	685x	916x
miniAero	Flat Plate	1,811x	—	—
NAS	IS Class S	204x	313x	294x
NAS	EP Class S	396x	542x	533x
NAS	CG Class S	12,169x	3,537x	3,855x
NAS	CG Class A	3,900x	—	—
NAS	MG Class S	5,163x	5,543x	3,129x
NAS	LU Class S	10,773x	10,080x	11,443x
Enzo	Cosmology Sim.	1,976x	—	—

Figure 12: Summary of Benchmarks.

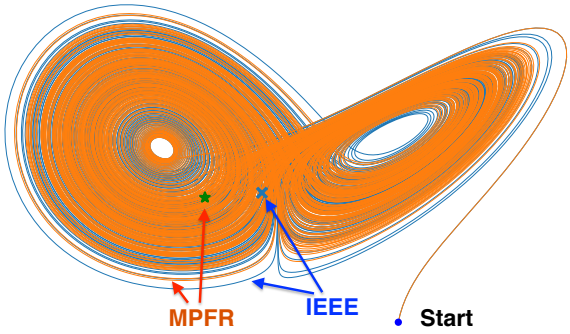

Figure 13: Lorenz system using IEEE and MPFR via FPVM. Blue trajectory is IEEE. Orange trajectory is MPFR. Trajectories and final state are different between the two arithmetic systems.

FPVM when a special event occurs. If an instruction uses non-NaN values and its result was not rounded, and did not overflow or underflow, FPVM is not invoked. The machine 7220 is a Dell 7720 with an Intel Xeon E3-1505M v6 and 32 GB of RAM running Ubuntu 20.04 with a stock 5.4.0 kernel. The machine R730xd is a Dell R730xd with two Intel Xeon E5-2695 v3 processors and 220 GB of RAM running RHEL 8.5 with a 4.18.0 kernel. Slowdowns are similar on these newer machines, although CG.S is an outlier.

5.4 Effects

Figure 13 shows the output of running a Lorenz system simulation for 2500 time steps under the hybrid FPVM prototype using three different arithmetic systems (original IEEE doubles, IEEE doubles emulated via FPVM (Vanilla), and MPFR emulated via FPVM) plugged in. Simply adding the FPVM layer, and thus trapping and emulating all floating point instructions that round, does not change the answer. There is no difference between the original IEEE doubles and the emulated IEEE doubles. On the other hand, using MPFR, with a higher precision, does indeed change the answer, as expected. Given a common starting point, the trajectories of IEEE and MPFR soon diverge, and this divergence is reflected in the final state (position).

What is happening here is that the Lorenz system is the classic example of a chaotic dynamic system [45]. As such, tiny changes in the initial condition, or tiny perturbations in intermediate states result in a divergence, typically an exponential divergence, over time. Each rounding event in the computation is such a perturbation. The rounding events encountered by IEEE and MPFR are different, resulting in the different trajectories and ending points.

5.5 Software engineering complexity

Part of the feasibility of FPVM stems from the software engineering effort and time spent building it. The hybrid FPVM outlined in Section 4 was comprised of roughly 6300 lines of C and C++ for the trap-and-emulate component, and 1484 lines of Python for the static analyzer. Individually, each alternative math binding was roughly 350 lines of code, leading us to believe that extending FPVM to support new alternative arithmetic is relatively simple.

5.6 Beyond x64

The design and implementation of our prototype is tightly coupled to x64. Is it portable to other CPU platforms, such as ARM and RISC-V? Or to GPU platforms?

A core requirement of our prototype (as well as the trap-and-emulate, and trap-and-patch approaches in general) is that the hardware can convert detection of the floating point exceptions (e.g., overflow, underflow, rounding, invalids, etc) into traps/interrupts that are delivered to the underlying kernel and thus to FPVM. These traps/interrupts are what drive FPVM. The x64 standards guarantee this behavior. The ARM standards make this behavior optional. Some platforms we have tested (such as ThunderX2, A64FX, and Apple M1) do indeed have the needed behavior. Others (such as ThunderX1) do not have it. RISC-V, at least in its "F" and "D" extensions, explicitly do *not* support this behavior for performance reasons. This is a shame because the "N" extension would be very beneficial for FPVM, as we describe later.

Is the hybrid FPVM prototype portable to GPUs? It is our understanding that current NVIDIA GPUs support neither the floating point condition codes nor raising a trap/interrupt when a condition code is set. This was also true in the past for AMD and Intel GPUs, but AMD's Vega architecture has support for floating point condition codes and exceptions.

On platforms in which traps/interrupts on floating point events are not available, FPVM would likely need to be implemented using the static analysis and transformation approach or the compiler-based transform approach.

6 Prospects for reducing overhead

We now describe techniques for further reducing the costs and complexity of floating point virtualization. Although our presentation is geared to the Linux and x64 context, all the techniques could be applied to other environments provided they have the basic functionality of presenting IEEE floating point exceptions as traps. If they do not, a static binary transformation, or the compiler-based approach would be needed to introduce checks in software.

6.1 Kernel-level support

Recall that in the trap-and-emulate model there is no overhead unless a shadowed value is involved. These events are detected by a hardware floating point exception. The overhead of delivering such an exception dominates the virtualization overhead of the FPVM prototype because they must propagate all the way to the user-level implementation of FPVM.

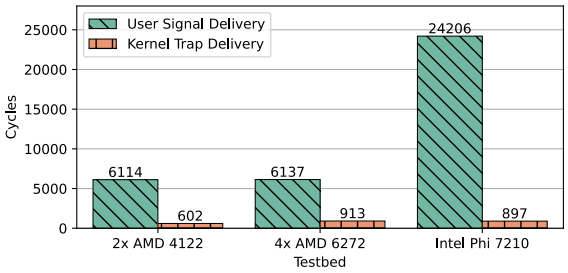

Figure 14: Overhead of user-level versus kernel-level exceptions/interrupts (quoted from [24]).

Figure 14 shows the measured overhead of delivering the exception to user-level and to kernel-level in three different platforms, including an AMD 6272-based machine similar to the one used for performance measurement in this paper. Kernel-level delivery has 7 to 30 times lower overhead. These numbers were measured on a kernel without Spectre/Meltdown mitigations and thus are likely to be conservative. One way to make use of these results would be to make FPVM a kernel-level service, implemented, for example, as a kernel module for Linux. This would reduce the overhead to be closer to that of kernel-level delivery, but we would still bear the cost of crossing the kernel-user boundary. A more aggressive option would be to incorporate FPVM into a pure-kernel execution model, such as in the hybrid run-time (HRT) model [27, 28]. By discarding kernel-user crossings altogether, the baseline overhead for FPVM would be similar to the "Kernel Trap Delivery" variants in Figure 14. Currently, only an implementation of FPVM as a Linux kernel module is planned.

6.2 Hardware support

Several small hardware changes would allow us to reduce or eliminate the need for static analysis, as well as to reduce the runtime overhead of FPVM-like trap-and-emulate virtualization of the floating point hardware.

Extending floating point traps for all x64 instructions: Our hybrid FPVM prototype uses expensive static binary analysis in order to handle edge cases in virtualizing the floating point hardware. One of these cases is straightforward: the x64 floating point hardware also includes support for integer and saturating arithmetic. As a result, it is possible, for example, for a NaN to flow into an logical operation like an XOR. While situations like this are rare, they do occur because these non-FP operations may be used by the compiler to optimize FP math. Currently, these situations do not result in an FP exception or trap, and thus our analysis and patching is necessary. The hardware could support a NaN input check for all operations, letting us avoid some of this analysis.

Trap on NaN-load: Similarly, a floating point value stored in memory might be treated as as an integer value in some circumstances, for example, from idiomatic C like `*(uint64_t *)(&x)` where `x` is a `double`. The majority of our static analysis is done to conservatively handle this kind of situation, forcing a trap into FPVM so it can determine if the value that is escaping as an integer is a NaN-boxed value.

NaNs, however, have bit patterns that make them unusual to be encountered in the wild. Furthermore, when running with FPVM, the majority of 8 byte quantities loaded that match a NaN pattern are likely to be NaN-boxed values created by FPVM. If the hardware could optionally trigger an exception when a NaN pattern is loaded as a value, the static analysis could be avoided.

User→user trap with fast delivery: As Figure 14 shows, the overhead of the delivery of a floating point exception as a trap is substantial, even ignoring kernel→user delivery. The hardware cost of delivery and return is on the order of 1000 cycles. In part, this high cost is due to the complex, stack-based exception/interrupt delivery mechanism on x64 and the need for a user→kernel→user privilege transition sequence. However, traps resulting from floating point operations do not require any of this—of the six traps available on x64 (Invalid, Inexact, Underflow, Overflow, Denorm, DivideByZero), only DivideByZero might have a consequence for the kernel or other processes. And DivideByZero applies only to integer operations (a floating point divide by zero is an Overflow).

A specialized exception delivery system geared to user→user privilege transfer similar to RISC-V's "N" extension [1] could dramatically lower the overhead of FPVM-like systems. For floating point virtualization, all that is needed is a same-privilege control flow transfer, which we anticipate could be brought down to less than 10 cycles on x64 through integration with branch prediction, a technique we refer to as a "pipeline interrupt". A proof-of-concept of this exists (in PIN). The interface is similar to the x64 `syscall` instruction, which already avoids stack operations for system calls, and to the TSX RTM transactional memory feature, which has a user→user transaction abort. Our scheme in effect dispatches the exception as a jump to a target address stored in an MSR[11] with the address of the faulting instruction and current flags placed in link registers implemented as MSRs.[12] We have measured TSX RTM transaction abort times as low as 100 cycles on current processors (Intel i7-9850H, specifically), and that includes unwinding the aborting transaction, which a user→user transfer would not do.

7 Conclusions

As promising alternative arithmetic representations emerge, providing higher precision than existing IEEE floats, they have not seen major adoption among the scientific community. This can mostly be attributed to the significant engineering effort required to such enormous codebases to support them. In this work, we explored possible approaches to address this challenge through classical VMM techniques by virtualizing the IEEE floating point hardware, specifically on x86. The goal of a floating point virtual machine (FPVM) is to allow existing application *binaries* to be transparently extended to support arbitrary alternative arithmetic systems without incurring significant virtualization overhead. We discussed the design and implementation of a prototype hybrid FPVM using trap-and-emulate and static value set analysis to evaluate the effects of alternative arithmetic on various scientific applications and benchmarks. We evaluated the overheads of said implementation, deficiencies of existing floating point hardware, as well as a few prospects to reduce virtualization overheads in future hardware.

[11] FPVM would load this with its entry point.
[12] FPVM would use these to resume execution after handling a delivered fault.

References

[1] The risc-v instruction set manual. volume i: User-level isa.

[2] Capstone: The ultimate disassembler, 2021.

[3] 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.

[4] 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.

[5] Balakrishnan, G., and Reps, T. Analyzing memory accesses in x86 executables. In *International conference on compiler construction* (2004), Springer, pp. 5–23.

[6] Bao, T., and Zhang, X. 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)* (October 2013).

[7] Bellard, F. Libbf: The tiny big float library. Available at https://bellard.org/libbf/, 2017.

[8] Bentley, M., Briggs, I., Gopalakrishnan, G., Ahn, D. H., Laguna, I., Lee, G. L., and Jones, H. E. 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)* (June 2019).

[9] Benz, F., Hildebrandt, A., and Hack, S. 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)* (2012).

[10] Boehm, H.-J. Simple garbage-collector-safety. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1996), PLDI '96, Association for Computing Machinery, p. 89–98.

[11] 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).

[12] 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.

[13] Cherkaev, A. The secret life of a nan. https://anniecherkaev.com/the-secret-life-of-nan, March 2018.

[14] Chiang, W.-F., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., and Rakamarić, Z. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)* (2017), pp. 300–315.

[15] Courbet, C. Nsan: A floating-point numerical sanitizer. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC)* (March 2021).

[16] Crozier, P., Thornquist, H., Numrich, R., Williams, A., Edwards, H., Keiter, E., Rajan, M., Willenbring, J., Doerfler, D., and Heroux, M. Improving performance via mini-applications. Tech. Rep. SAND2009-5574, Sandia National Laboratories, January 2009.

[17] Devine, S., Bugnion, E., and Rosenblum, M. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. United States Patent Number 6397242.

[18] Dinda, P., and Bernat, A. Comparing the understanding of ieee floating point between scientific and non-scientific users. Tech. Rep. NWU-CS-2021-07, Department of Computer Science, Northwestern University, December 2021.

[19] Dinda, P., Bernat, A., and Hetland, C. Spying on the floating point behavior of existing, unmodified scientific applications. In *Proceedings of the 29th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2020)* (June 2020). Best Paper.

[20] Dinda, P., and Hetland, C. Do developers understand IEEE floating point? In *Proceedings of the 32rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018)* (Apr. 2018).

[21] Duck, G. J., Gao, X., and Roychoudhury, A. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2020), PLDI 2020, Association for Computing Machinery, p. 151–163.

[22] Févotte, F., and Lathuilière, B. VERROU: assessing floating point accuracy without recompiling, October 2016. working paper or preprint.

[23] 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).

[24] Ghosh, S., Cuevas, M., Campanoni, S., and Dinda, P. Compiler-based timing for extremely fine-grain preemptive parallelism. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC 2020)* (November 2020).

[25] Goldberg, R. Survey of virtual machine research. *IEEE Computer* (June 1974), 34–45.

[26] Gustafson, J. *The End of Error: Unum Computing*. Chapman and Hall/CRC, 2015.

[27] Hale, K., and Dinda, P. A case for transforming parallel runtimes into operating system kernels. In *Proceedings of the 24th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2015)* (June 2015).

[28] Hale, K., and Dinda, P. Enabling hybrid parallel runtimes through kernel and virtualization support. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2016)* (April 2016).

[29] 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.

[30] Hollingsworth, J. K., and Buck, B. *DynInstAPI Programmer's Guide Release 1.0*, July 1997. http://www.cs.umd.edu/ hollings/dyninstAPI/dyninstUserGuide.pdf.

[31] Ian A. Mason, S. I. https://github.com/SRI-CSL/gllvm, 2018.

[32] IEEE Floating Point Working Group. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985* (1985).

[33] IEEE Floating Point Working Group. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70.

[34] Jin, H., Frumkin, M., and Yan, J. The openmp 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.

[35] 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).

[36] 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).

[37] 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).

[38] 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.

[39] Lam, M. O., Hollingsworth, J. K., and Stewart, G. Dynamic floating-point cancellation detection. *Parallel Computing 39*, 3 (2013), 146–155.

[40] Landi, W. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst. 1*, 4 (dec 1992), 323–337.

[41] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, Mar 2004).

[42] Lee, W.-C., Bao, T., Zheng, Y., Zhang, X., Vora, K., and Gupta, R. 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)* (2015).

[43] Matula, D. W., and Kornerup, P. Finite precision rational arithmetic: Slash number systems. *IEEE Transactions on Computers C-34*, 1 (Jan 1985), 3–18.

[44] Milroy, D. J., Baker, A. H., Hammerling, D. M., Dennis, J. M., Mickelson, S. A., and Jessup, E. R. Towards characterizing the variability of statistically consistent community earth system model simulations. *Procedia Computer Science 80*, C (June 2016), 1589–1600.

[45] Moon, F. C. *Chaotic and Fractal Dynamics: An Introduction for Applied Scientists and Engineers*. John Wiley and Sons, Inc., 1992.

[46] 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.

[47] Omtzigt, E. T. L., Gottschling, P., Seligman, M., and Zorn, W. Universal Numbers Library: design and implementation of a high-performance reproducible number systems library. *arXiv:2012.11011* (2020).

[48] Panchekha, P., Sanchez-Stern, A., Wilcox, J. R., and Tatlock, Z. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2015).

[49] Popek, G., and Goldberg, R. Formal requirements for virtualizable third generation architectures. *Communications of the ACM* (July 1974), 413–421.

[50] Ramalingam, G. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst. 16*, 5 (sep 1994), 1467–1471.

[51] Ravitch, T. https://github.com/travitch/whole-program-llvm, 2016.

[52] 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).

[53] Sanchez-Stern, A., Panchekha, P., Lerner, S., and Tatlock, Z. Finding root causes of floating point error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2018).

[54] Sawaya, G., Bentley, M., Briggs, I., Gopalakrishnan, G., and Ahn, D. H. Flit: Cross-platform floating-point result-consistency tester and workload. In *Proceedings of the 2017 IEEE International Symposium on Workload Characterization (IISWC)* (Oct 2017), pp. 229–238.

[55] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. Sok: (state of) the art of war: Offensive techniques in binary analysis.

[56] Sugerman, J., Venkitachalan, G., and Lim, B.-H. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference* (June 2001).

[57] Walker, J. Fbench: Floating point benchmarks. https://www.fourmilab.ch/fbench/, September 2021.

[58] Wingo, A. Value representation in javascript implementations. http://wingolog.org/archives/2011/05/18/value-representation-in-javascript-implementations, May 2011.