Virtualization So Light, it *Floats*! Accelerating Floating Point Virtualization

Nick Wanninger, Nadharm Dhiantravan, Peter Dinda

Northwestern | Plab

Virtualization So Light, it *Floats*!
Accelerating Floating Point Virtualization

Nick Wanninger, Nadharm Dhiantravan, Peter Dinda

Northwestern | Plab

There are several alternatives to Floating Point

- AI Model quantization: float8, bfloat16, etc.
- Posit/Unum, rationals, arbitrary precision floating point, Bfloats, logarithmic arithmetic, ...
- ***A whole conference dedicated to this***

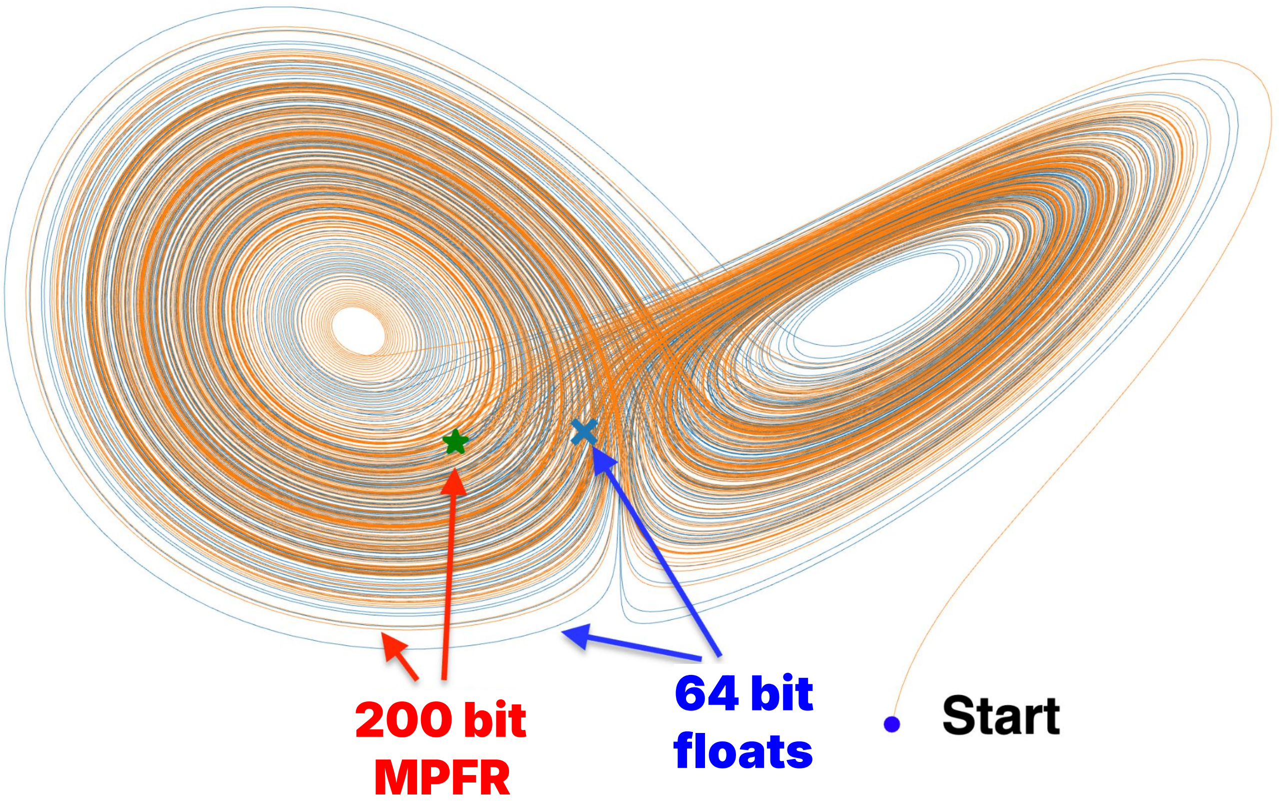
32nd IEEE International Symposium on Computer Arithmetic

ARITH 2025

El Paso, TX, USA. May 4-7, 2025.

https://www.arith2025.org/

Changing number systems *will* changes results.



Switching to these systems is nontrivial

```
double op(float a, float b, float c) {
    return a * b + c;
}
```

Switching to these systems is nontrivial

```
double op(float a, float b, float c) {
    return a * b + c;
}
```

```
void mpfr_op(mpfr_t result, mpfr_t a, mpfr_t b, mpfr_t c) {
    mpfr_mul(result, a, b, MPFR_RNDN); // result = a * b
    mpfr_add(result, result, c, MPFR_RNDN); // result += c
}
```

The entire code structure needs to change!

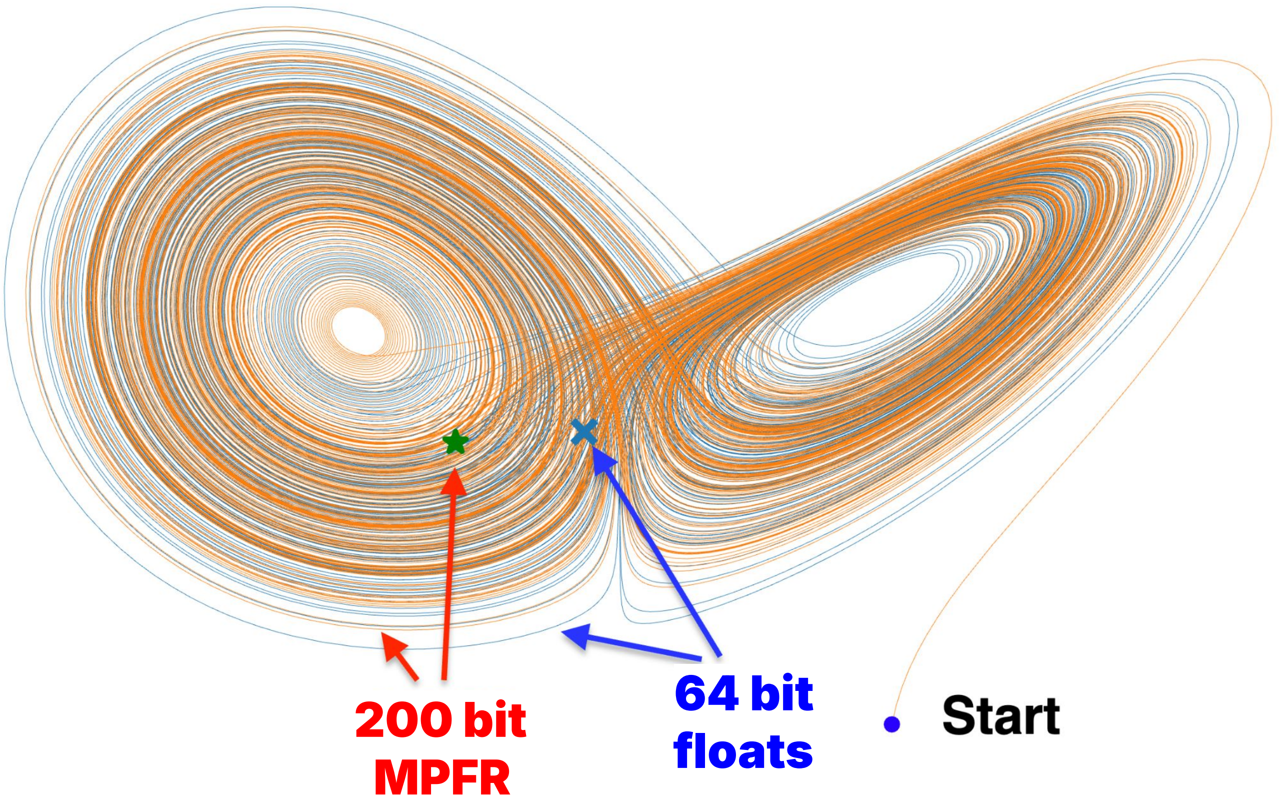
```
double op(float a, float b, float c) {
    return a * b + c;
}
```

Manually manage memory lifetimes of your numbers!

```
void mpfr_op(mpfr_t result, mpfr_t a, mpfr_t b, mpfr_t c) {
    mpfr_mul(result, a, b, MPFR_RNDN); // result = a * b
    mpfr_add(result, result, c, MPFR_RNDN); // result += c
}
```

Imagine needing to worry about this in something like CESM!

We want scientists to be able to experiment with these things



We want to *write* applications with the semantics of hardware floating point

But have it *execute* using some alternative arithmetic!

Floating Point Virtualization

- Have the program *think* it is using hardware floating point
- But swap it out, transparently through **virtualization**

(HPDC'22)

nickw.io/papers/hpdc22.pdf

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

A user can execute their ***"blessed binary"*** under FPVM simply:

```
$ fpvm run ./solve_climate_change input.csv
```

Without recompiling

FPVM is a Virtual Machine

- No **hardware support** for virtualized floating point
- So we simulate it using **software**

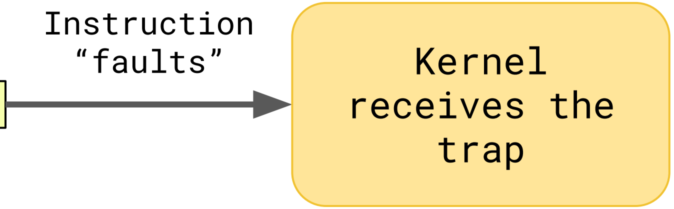
- Configure the hardware to **trap** when rounding, overflow, etc., occur.
- **Emulate** the instruction in software with a different arithmetic system

Let's say we have an instruction which rounds

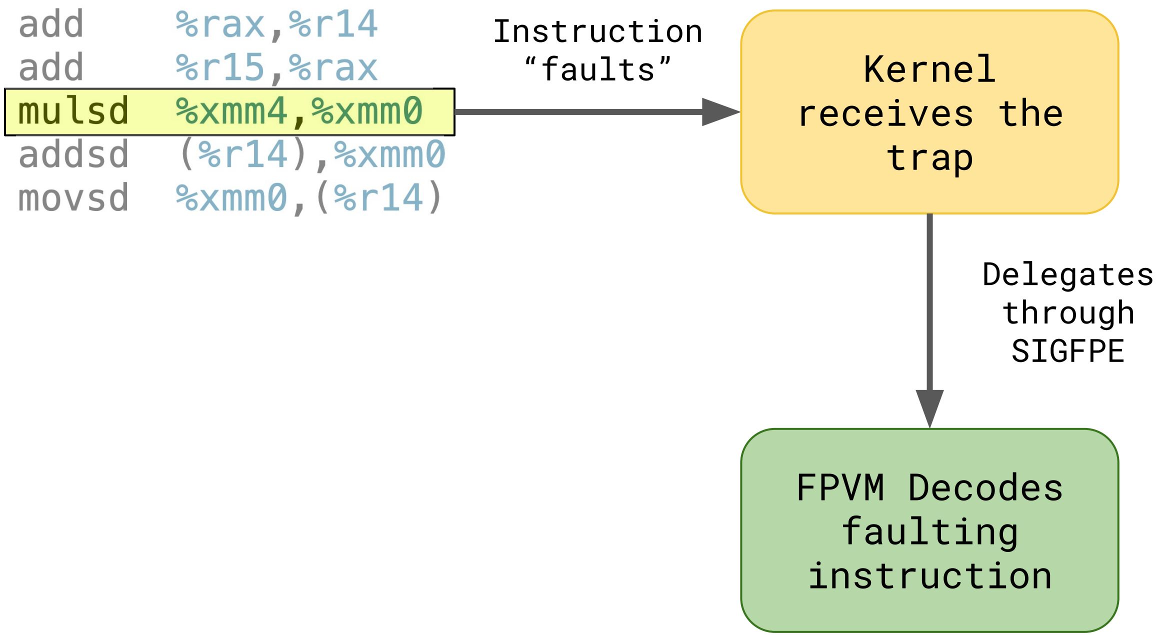
```
add    %rax,%r14
add    %r15,%rax
mulsd  %xmm4,%xmm0
addsd  (%r14),%xmm0
movsd  %xmm0,(%r14)
```

The hardware catches this and tells the kernel

```
add    %rax,%r14
add    %r15,%rax
mulsd  %xmm4,%xmm0
addsd  (%r14),%xmm0
movsd  %xmm0,(%r14)
```

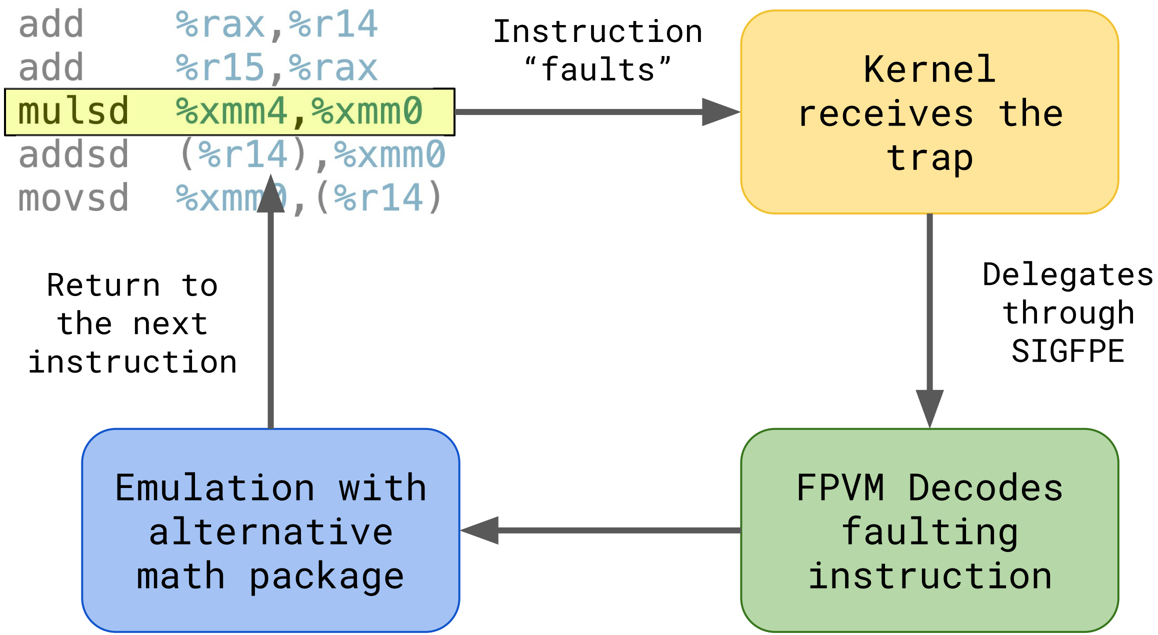


… which delegates the fault to FPVM with SIGFPE

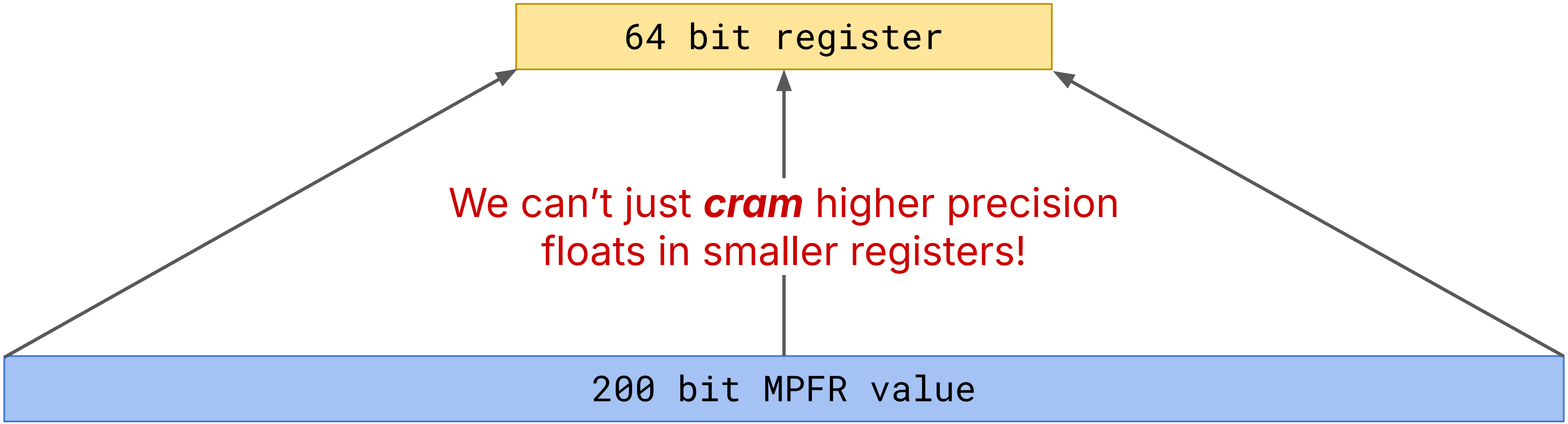


FPVM then emulates this instruction at a higher precision

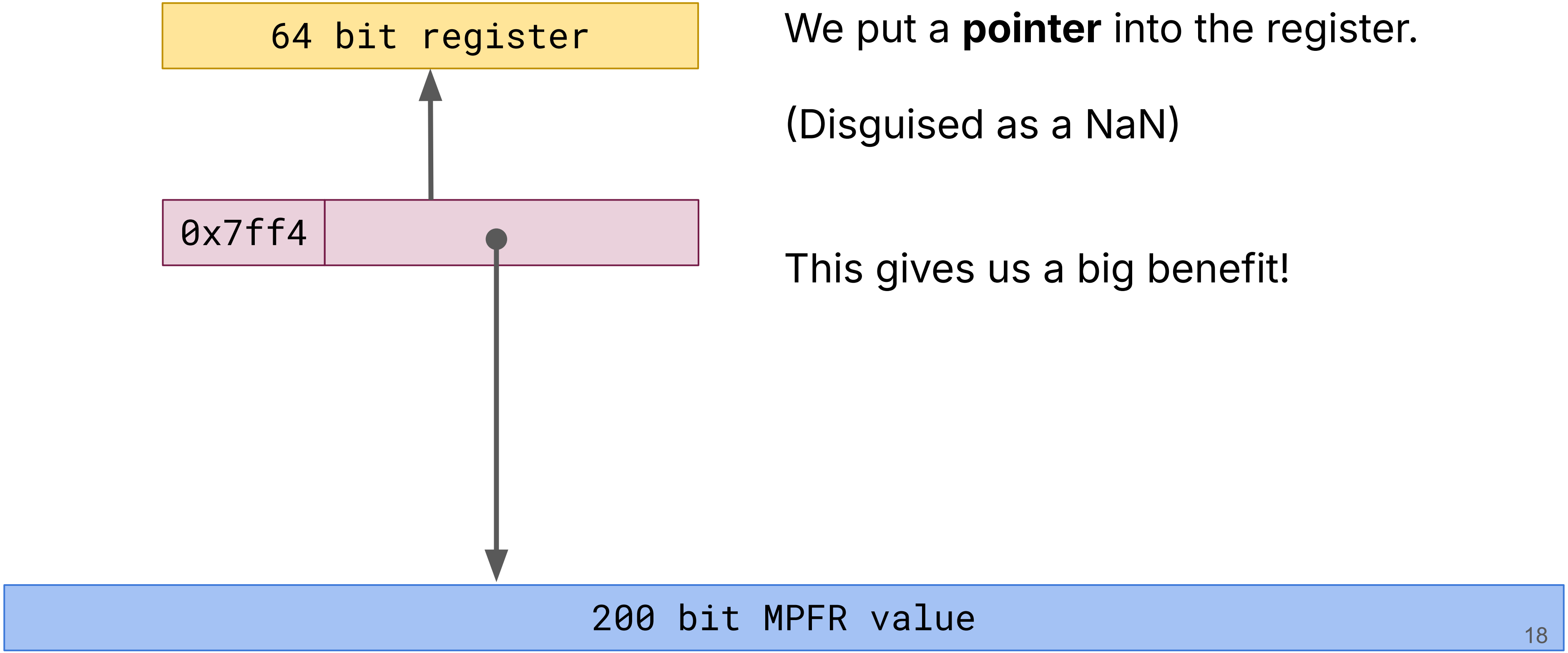
(e.g., 200 bit MPFR)



There's one problem with this…



Solution: NaN boxing

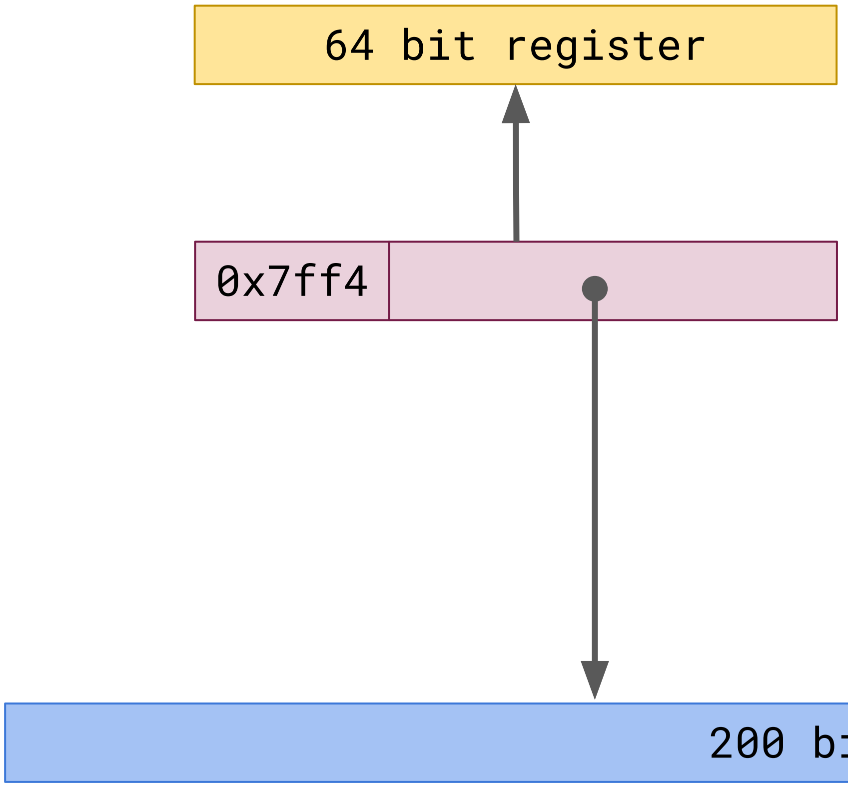


We put a **pointer** into the register.

(Disguised as a NaN)

This gives us a big benefit!

Solution: NaN boxing

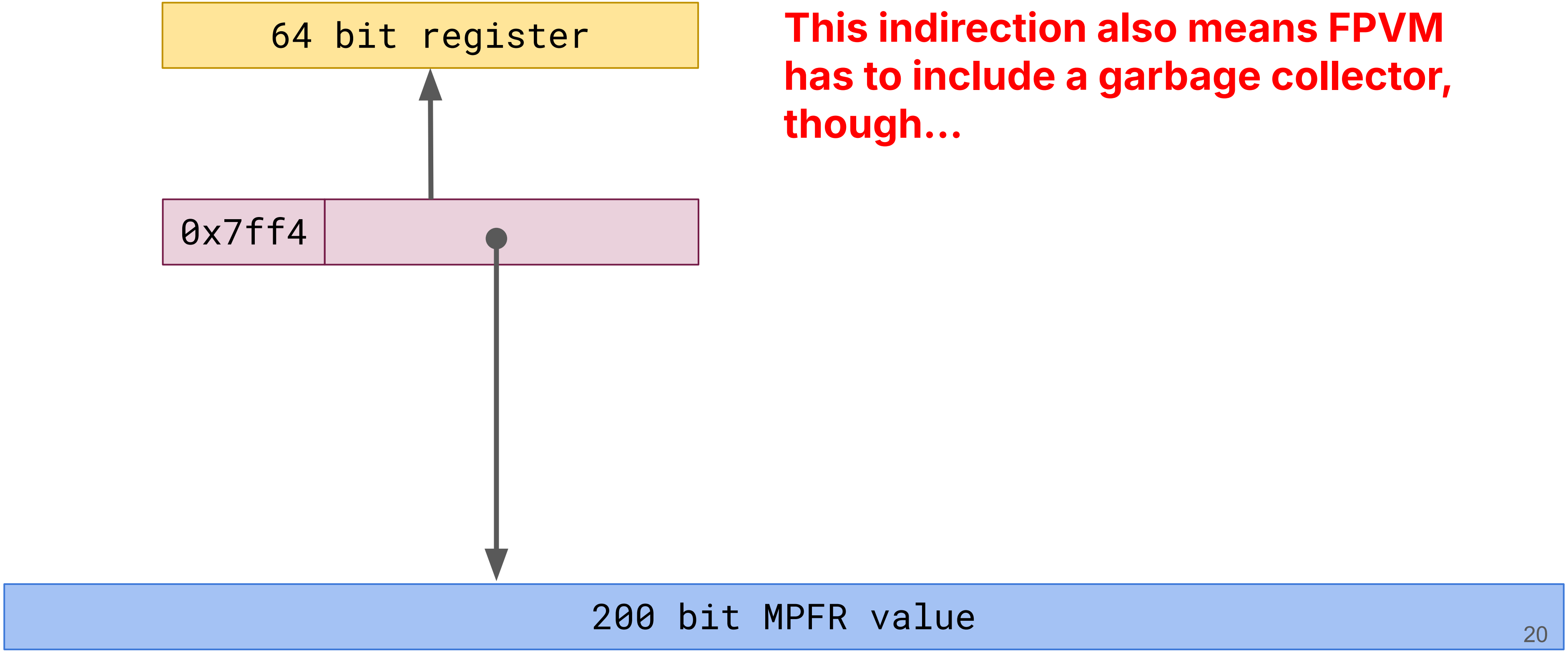


We put a **pointer** into the register.

(Disguised as a NaN)

Future accesses to this value will also trap into FPVM!

Solution: NaN boxing



This indirection also means FPVM has to include a garbage collector, though...

FPVM Supports four alternative arithmetic systems

Vanilla

Evaluate using IEEE Floating point hardware

Boxed

Vanilla, but with *NaN boxed* values

MPFR

Use arbitrary precision floats from the MPFR library

Posits

Experimental bindings to the posits alternative arithmetic system

These are broken down into two groups

Vanilla	Boxed	MPFR	Posits
Evaluate using IEEE Floating point hardware	Vanilla, but with *NaN boxed* values	Use arbitrary precision floats from the MPFR library	Experimental bindings to the posits alternative arithmetic system

Correctness Validation

Real *alternatives to IEEE floating point*

We'll focus on *Boxed* in this talk

Vanilla

Evaluate using IEEE Floating point hardware

Boxed

Vanilla, but with *NaN boxed* values

MPFR

Use arbitrary precision floats from the MPFR library

Posits

Experimental bindings to the posits alternative arithmetic system

Boxed is a minimal system that amplifies virtualization overhead

Unfortunately,

x86 is not fully floating point virtualizable.

We aren't going to get traps for **all** operations which should to maintain correctness.

Unfortunately,

x86 is not fully floating point virtualizable.

We aren't going to get traps for **all** operations which should to maintain correctness.

```
double x = ...;
long   y = *(long*)&x;
```

Treating floats as ints won't act right with NaNs

Unfortunately,

x86 is not fully floating point virtualizable.

We aren't going to get traps for **all** operations which should to maintain correctness.

```
double x = ...;
long   y = *(long*)&x;
```

Treating floats as ints won't act right with NaNs

```
double x = ...;
double z = -x;

movsd    ..., %xmm0
xorpd    %xmm1, (1 << 63)
```

The evil compiler thinks its *clever*...

Binary code analysis to the rescue!

```c
extern double fp;
int foo (double fp) {
return *(int*) &fp;
}
```

```asm
foo:
push rbp
mov rbp, rsp
movsd QWORD PTR [rbp-8], xmm0
lea rax, [rbp-8]
mov eax, DWORD PTR [rax]
pop rbp
ret
```

FPVM featured a binary analysis to *find these situations*

It then inserts "correctness traps"

```c
extern double fp;
int foo (double fp) {
return *(int*) &fp;
}
```

```asm
foo:
push rbp
mov rbp, rsp
movsd QWORD PTR [rbp-8], xmm0
lea rax, [rbp-8]
mov eax, DWORD PTR [rax]
pop rbp
ret
```

A trap to FPVM would be inserted here to *"demote"* eax back to a float

This work:

Virtualization So Light, it *Floats*! Accelerating Floating Point Virtualization

Nick Wanninger, Nadharm Dhiantravan, Peter Dinda

Northwestern | Plab

Virtualization So Light, it *Floats*!

Accelerating Floating Point Virtualization

Nick Wanninger, Nadharm Dhiantravan, Peter Dinda

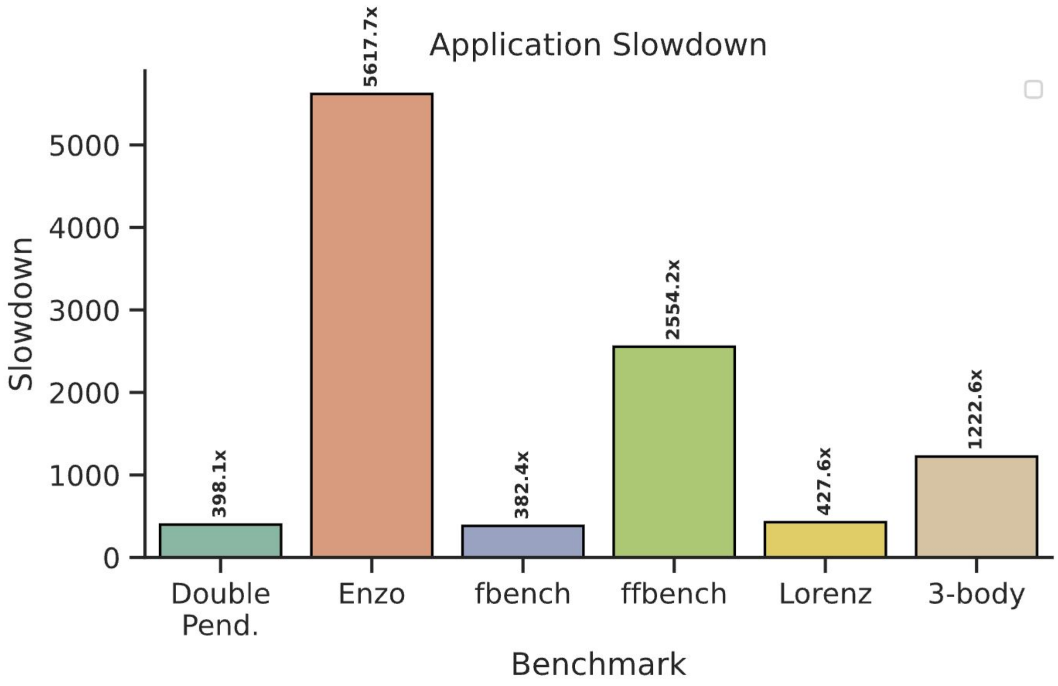
Northwestern | PLab

FPVM's performance has left room for improvement.

It enabled transparent swapping of arithmetic systems

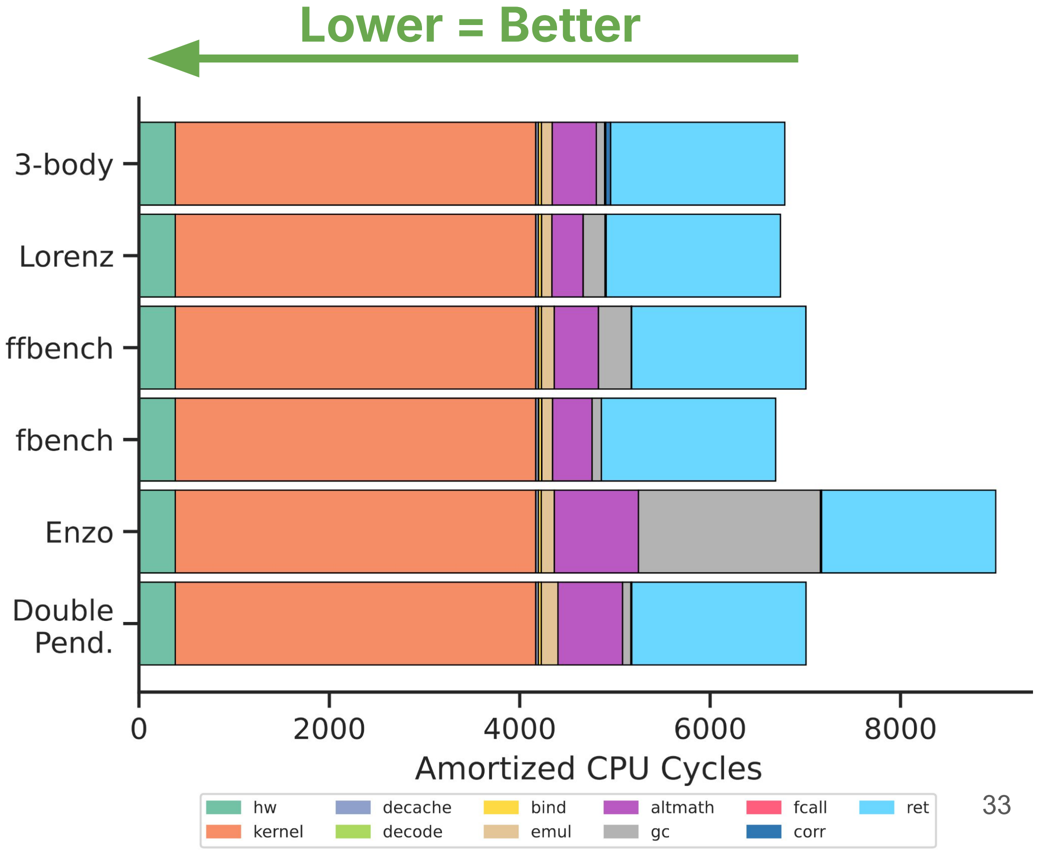
But... some applications had **6,000x slowdown**

Our baseline performance overheads

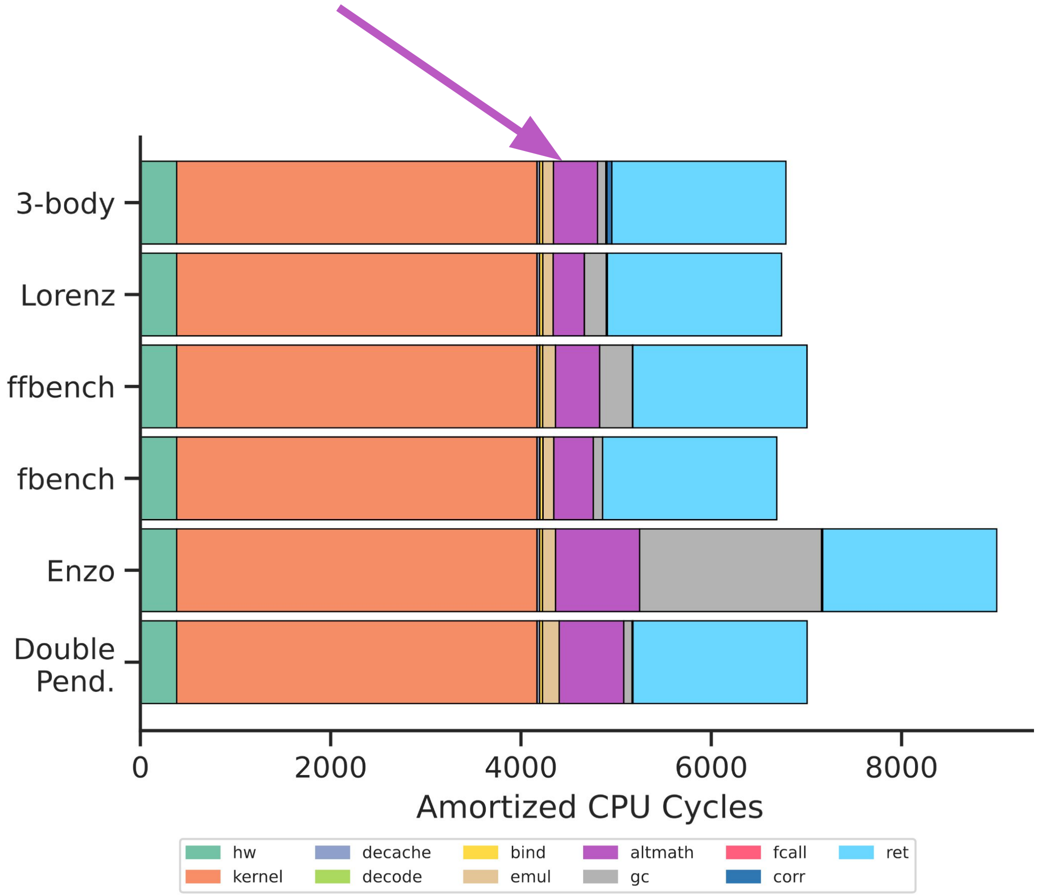


Breaking down the virtualization overhead

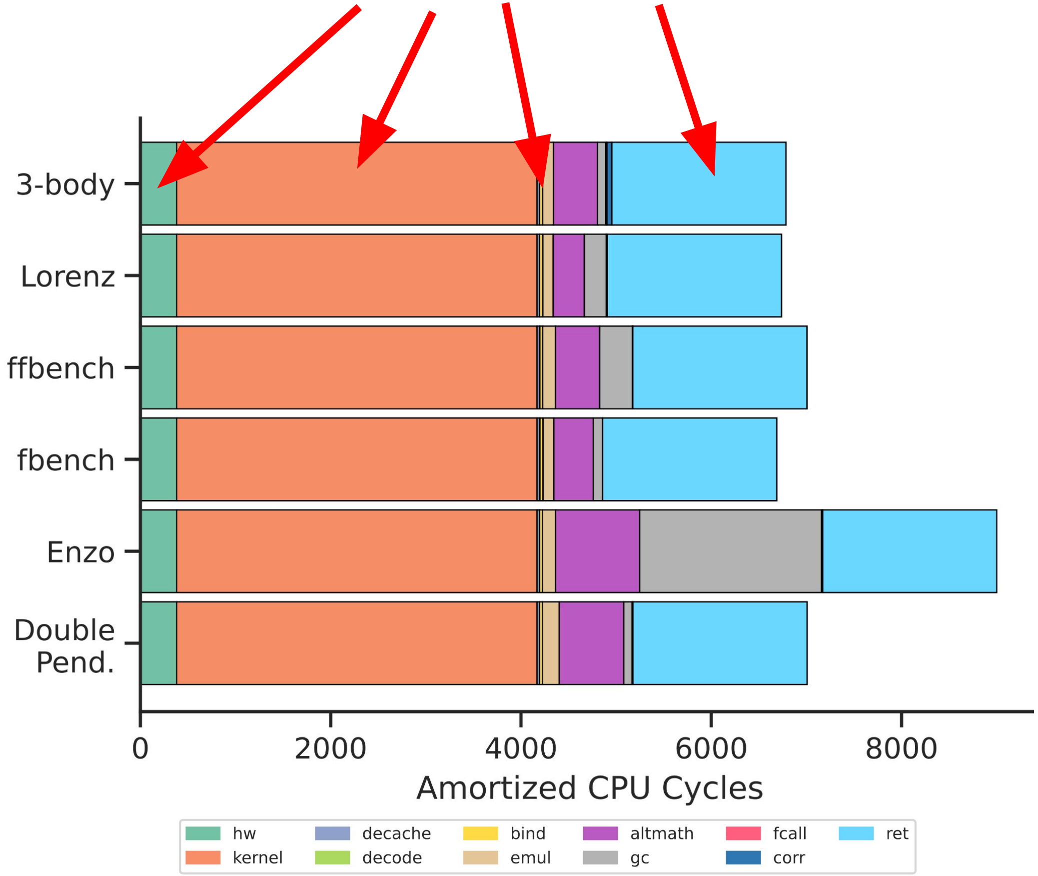
A instruction, the majority of the overhead comes from **signal delivery** and **returning to the next instruction**



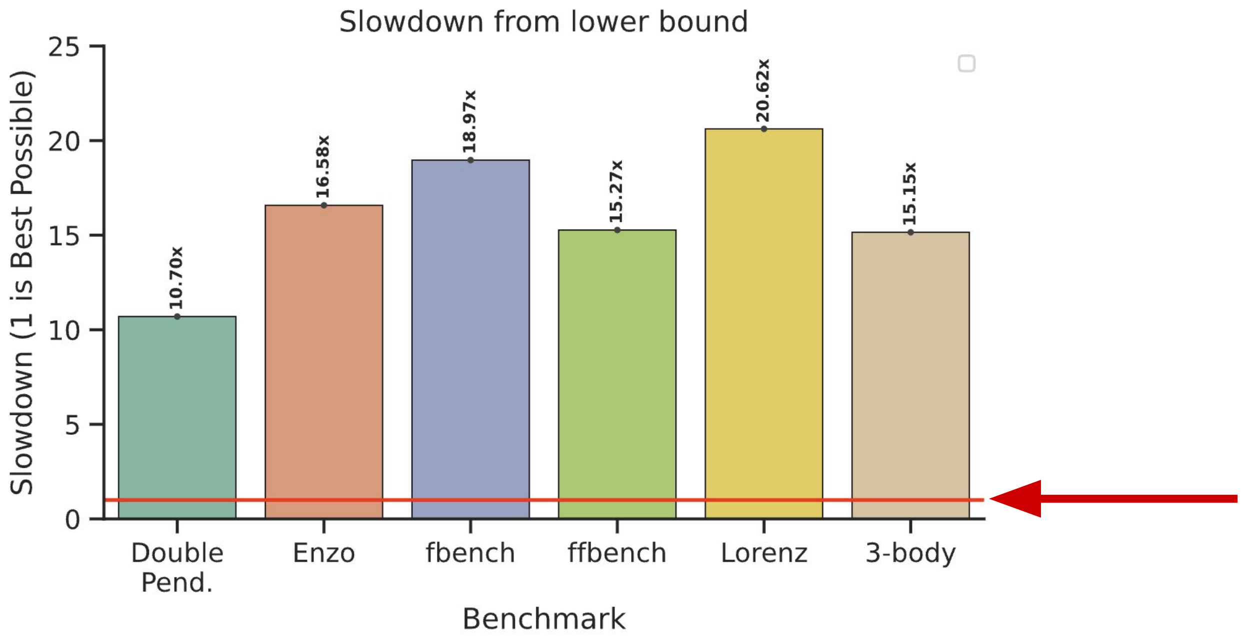
Ideally alternative math would be the *only* overhead



Everything else is virtualization overhead



FPVM was between 10 and 20x slower than our goal of zero-cost virtualization



1x is "zero virtualization overhead"

The goal of this paper is to get the *cost of virtualization* down to zero.

We do this with three techniques

Trap Short Circuiting

Sequence Emulation

Profiler based correctness traps

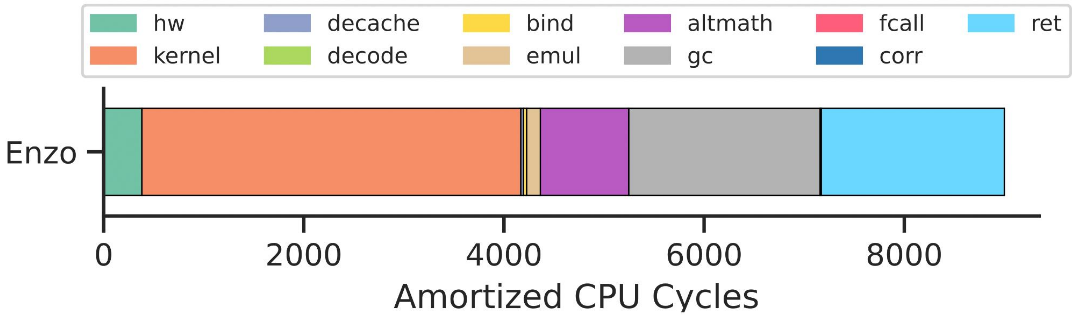
Trap short circuiting first

Trap Short Circuiting

Sequence Emulation

Profiler based correctness traps

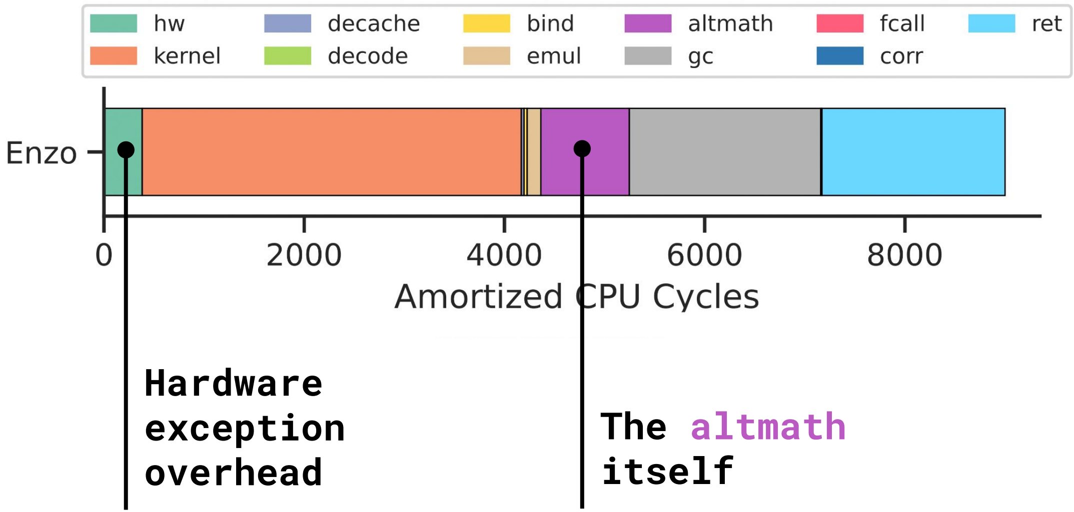
Let's take a closer look at the overheads



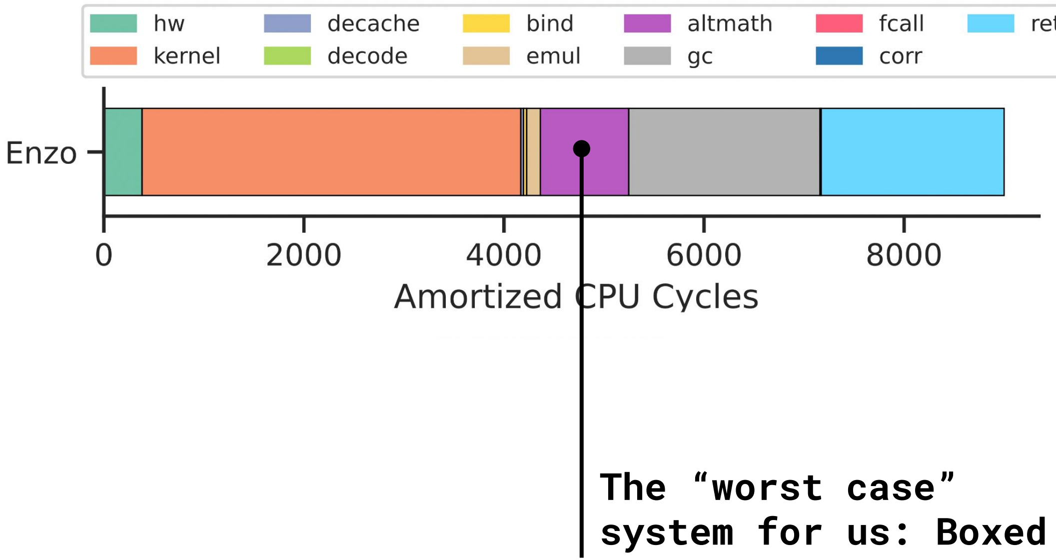
This is a non-trivial, large, multi-physics hydrodynamic astrophysical application

https://enzo-project.org/

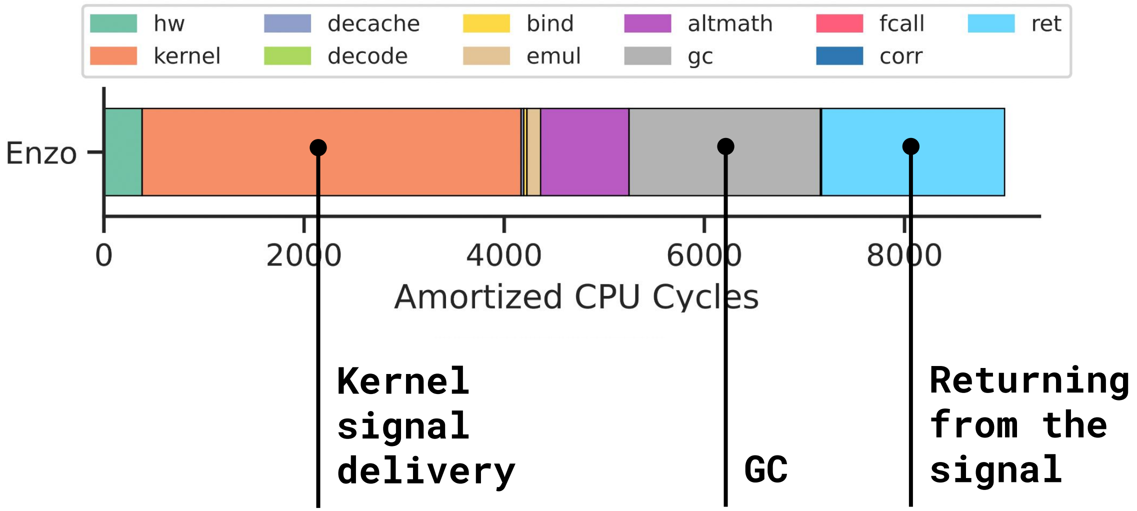
We have a few intrinsic overheads



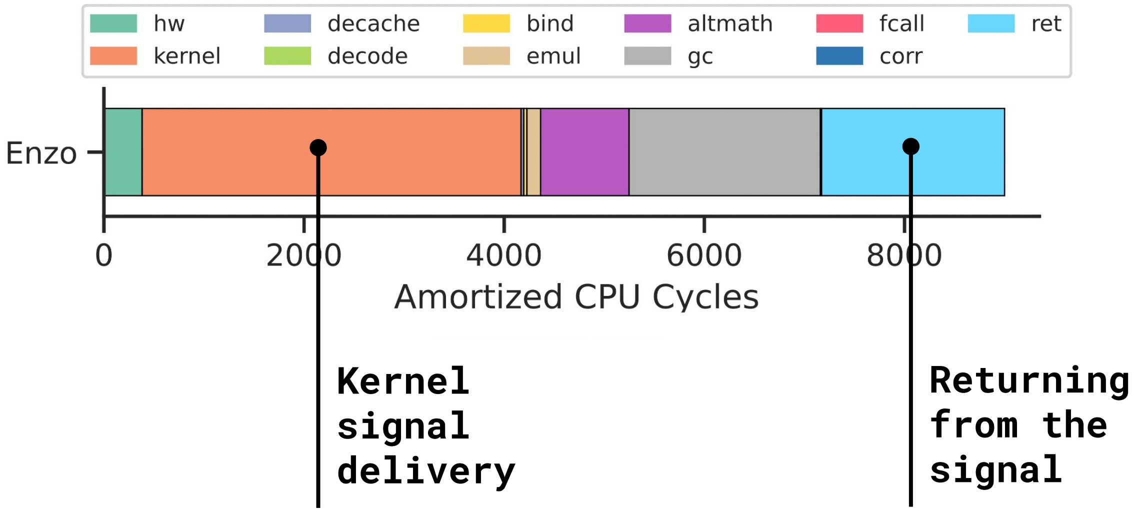
This test uses the minimum overhead altmath



But a few of these are solvable software problems



In this work, we'll focus on the signal overheads



Let’s attack the problem head on

- The FPVM runtime needs to be notified of floating point exceptions
- Existing signal mechanisms are designed to be general purpose, and relatively rare
- ... and as a result, are not as fast as they could be.

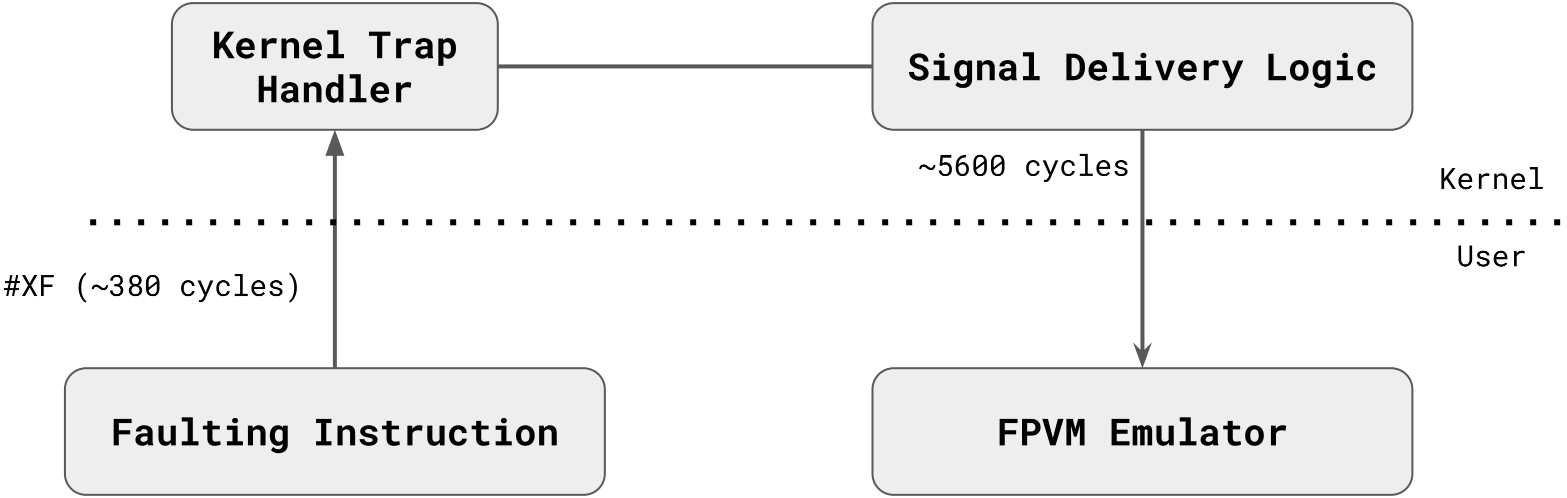
Let’s attack the problem head on

- The FPVM runtime needs to be notified of floating point exceptions
- Existing signal mechanisms are designed to be general purpose, and relatively rare
- ... and as a result, are not as fast as they could be.

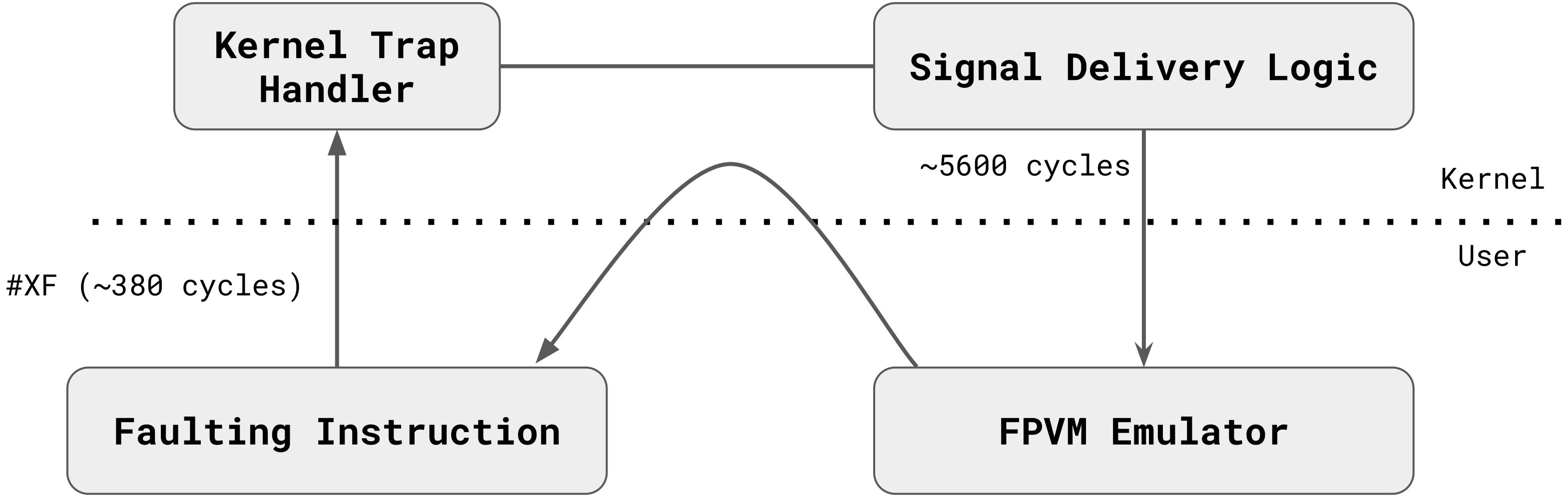
So let’s just replace signals!

Regular signal delivery is expensive

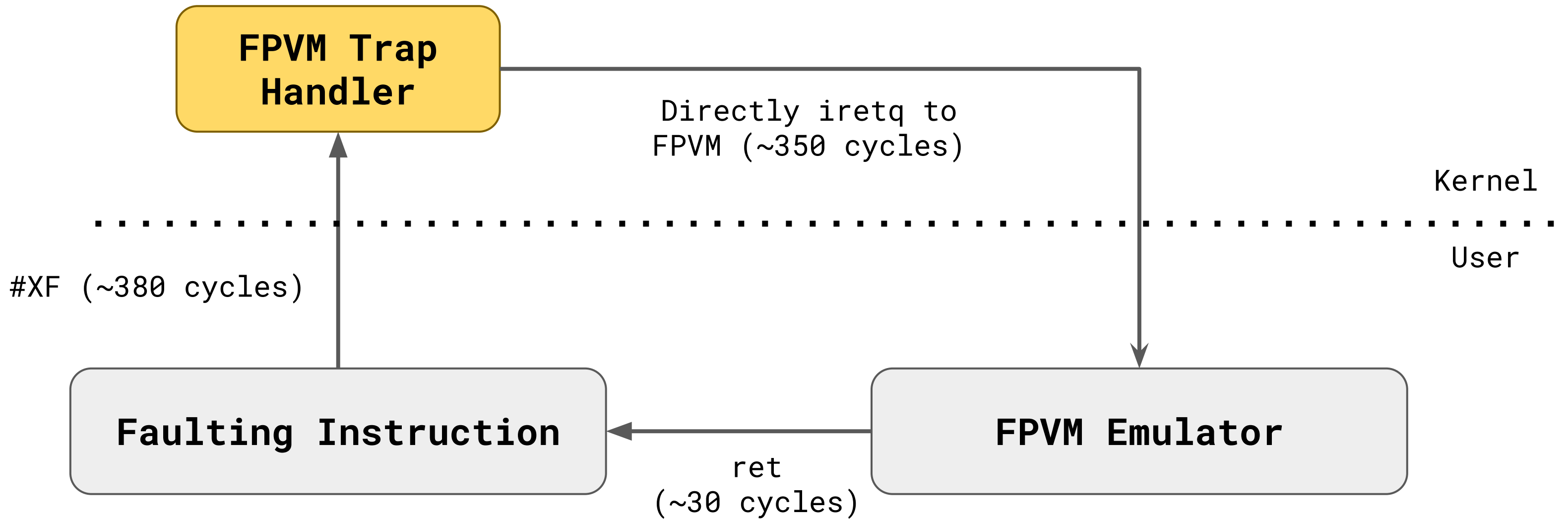
Regular signal delivery is expensive



Sigreturn is also slow!

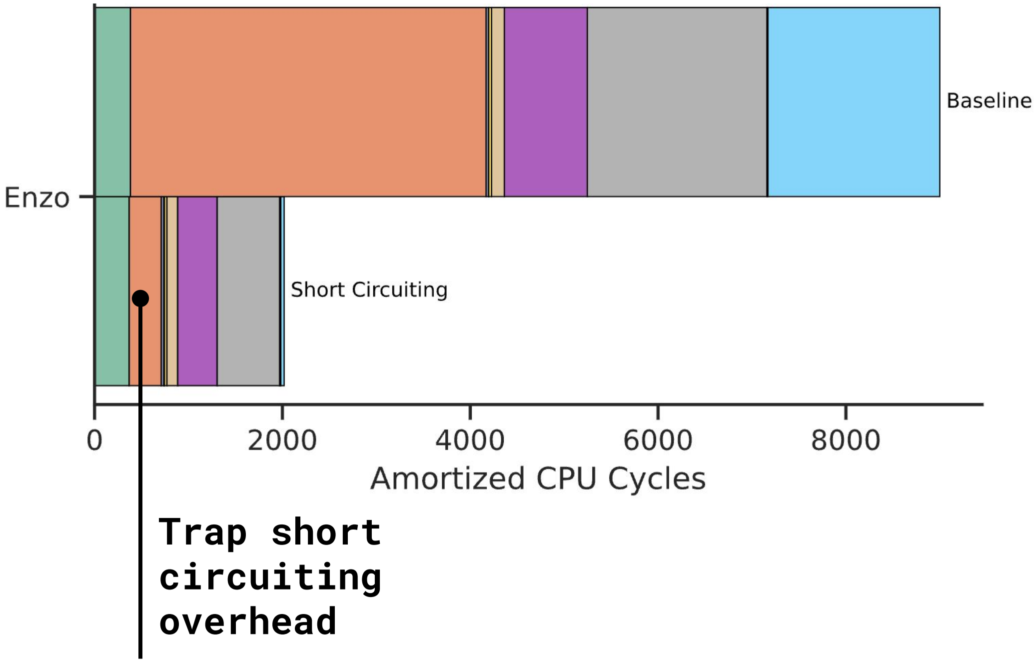


Trap Short Circuiting bypasses the signals

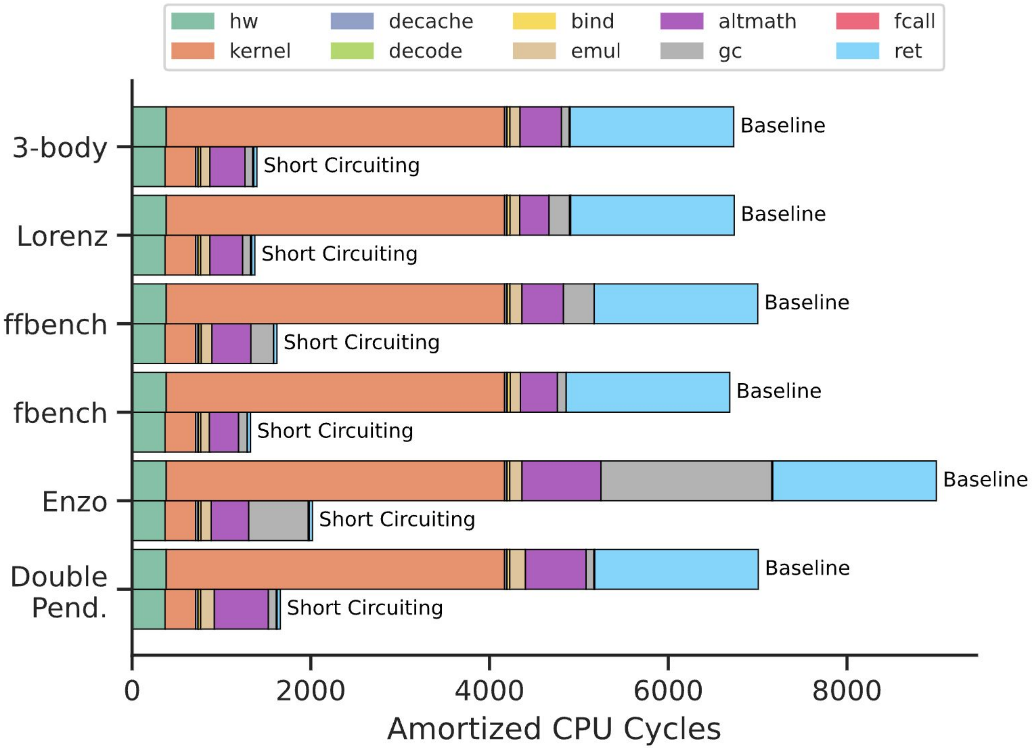


Trap short circuiting reduces overheads ***substantially***

- Kernel time is reduced by over **10x**
- It's now basically free to return from FPVM
- Overall overheads drop by **~6x**



This improvement is consistent



There’s more we can do, though.

Trap Short Circuiting

Sequence Emulation

Profiler based correctness traps

```
addsd   %xmm0, %xmm1
mulsd   %xmm0, %xmm0
divsd   %xmm0, %xmm2
```

FPVM emulation tends to cascade

```
addsd   %xmm0, %xmm1
mulsd   %xmm0, %xmm0
divsd   %xmm0, %xmm2
```

If this instruction traps

FPVM emulation tends to cascade

```
addsd    %xmm0, %xmm1
mulsd    %xmm0, %xmm0
divsd    %xmm0, %xmm2
```

So will this one

Sequence emulation amortizes overheads across instructions

```
addsd    %xmm0, %xmm1
mulsd    %xmm0, %xmm0
divsd    %xmm0, %xmm2
```

Trap!

Sequence emulation amortizes overheads across basic blocks

```
addsd    %xmm0, %xmm1
mulsd    %xmm0, %xmm0
divsd    %xmm0, %xmm2

           ⋮
```

We emulate all of these!

Sequence emulation amortizes overheads across instructions

```
addsd    %xmm0, %xmm1
mulsd    %xmm0, %xmm0
divsd    %xmm0, %xmm2

         ⋮
```

So we only pay exception handling once!

We have to be careful though!

```
addsd   %xmm0, %xmm1
mulsd   %xmm0, %xmm0
divsd   %xmm0, %xmm2
movsd   (...), %xmm2
addsd   %xmm0, %xmm2
```

We have to be careful though!

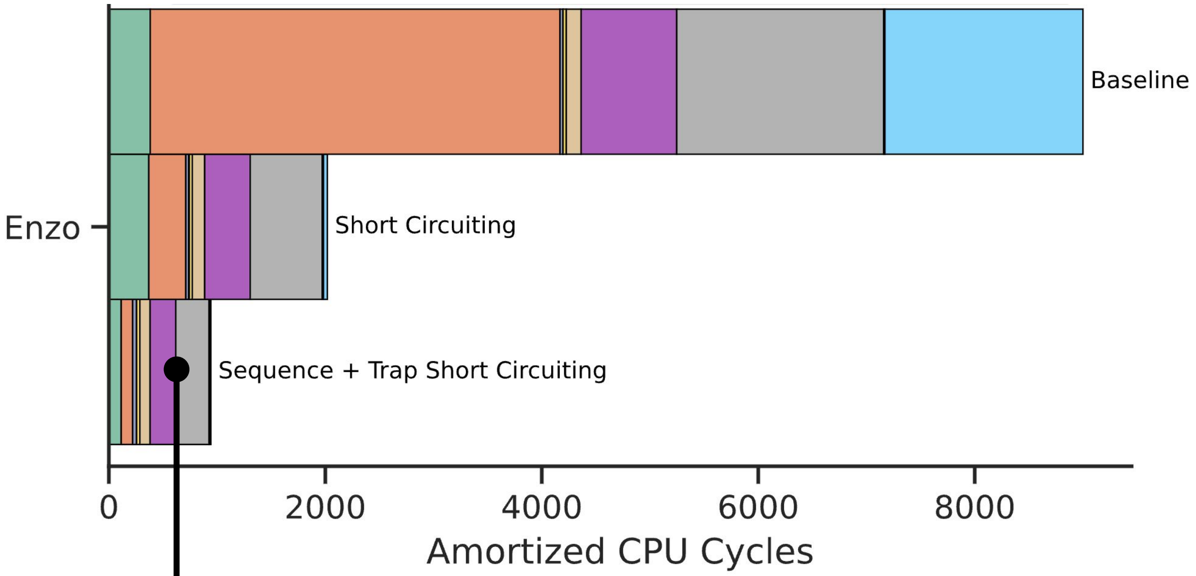
```
addsd    %xmm0, %xmm1
mulsd    %xmm0, %xmm0
divsd    %xmm0, %xmm2
movsd    (...), %xmm2
addsd    %xmm0, %xmm2
```

Most FP sequences are broken up by a few NON-FP instructions!

We extended FPVM to emulate these instructions

```
addsd    %xmm0, %xmm1
mulsd    %xmm0, %xmm0
divsd    %xmm0, %xmm2
movsd    (...), %xmm2
addsd    %xmm0, %xmm2
```

Combining these solutions nearly eliminates kernel overhead



Overhead now dominated by altmath and GC

Very quickly, our last technique…

Trap Short Circuiting

Sequence Emulation

Profiler based correctness traps

This technique attacks the *User Experience*

The previous technique to insert correctness traps could take ***weeks*** to complete.

```c
extern double fp;
int foo (double fp) {
return *(int*) &fp;
}
```

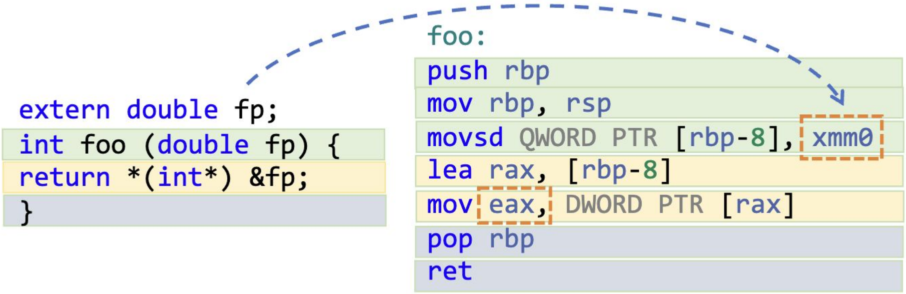
```asm
foo:
push rbp
mov rbp, rsp
movsd QWORD PTR [rbp-8], xmm0
lea rax, [rbp-8]
mov eax, DWORD PTR [rax]
pop rbp
ret
```

This is because it attempts to solve an ***unsolvable problem***

(alias analysis)

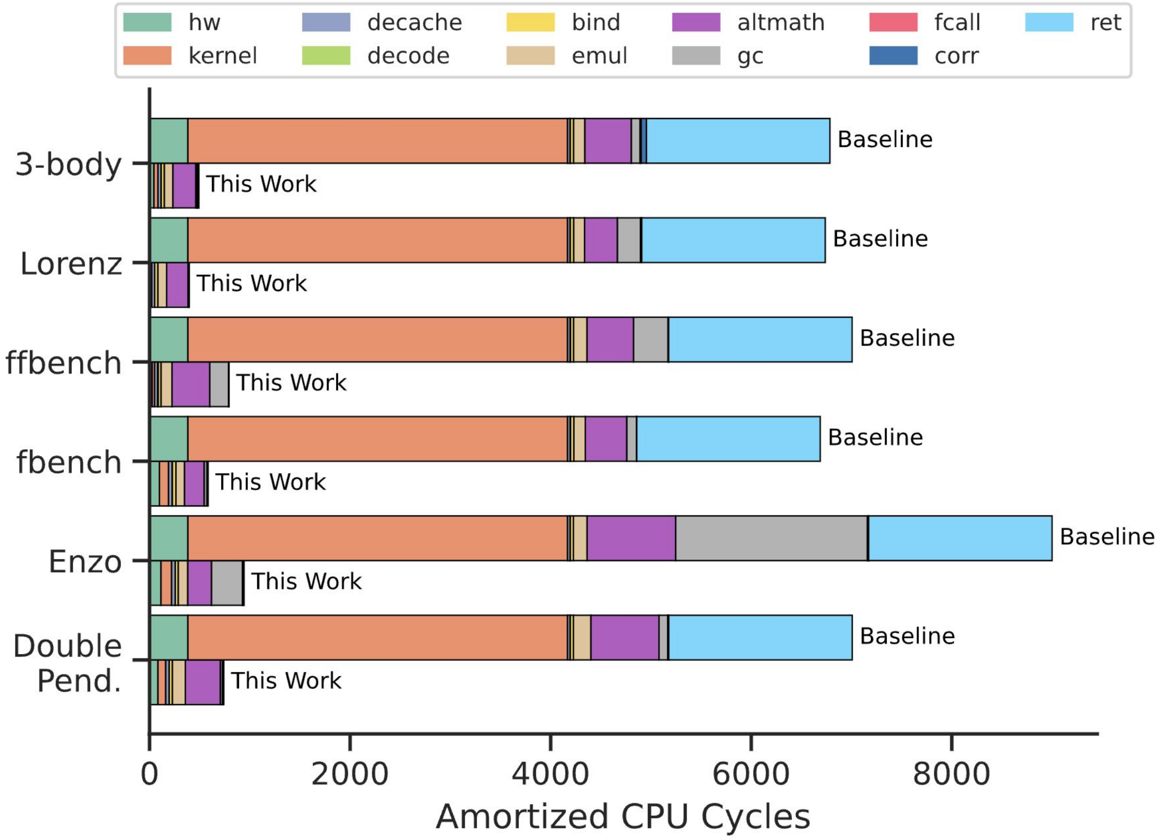
We replaced this analysis with a *profiler*

- Run your program *once* through a profiler
- "Representative workload"
- Analysis times down from ***weeks*** to ***minutes***
- **FPVM can now run many more programs!**

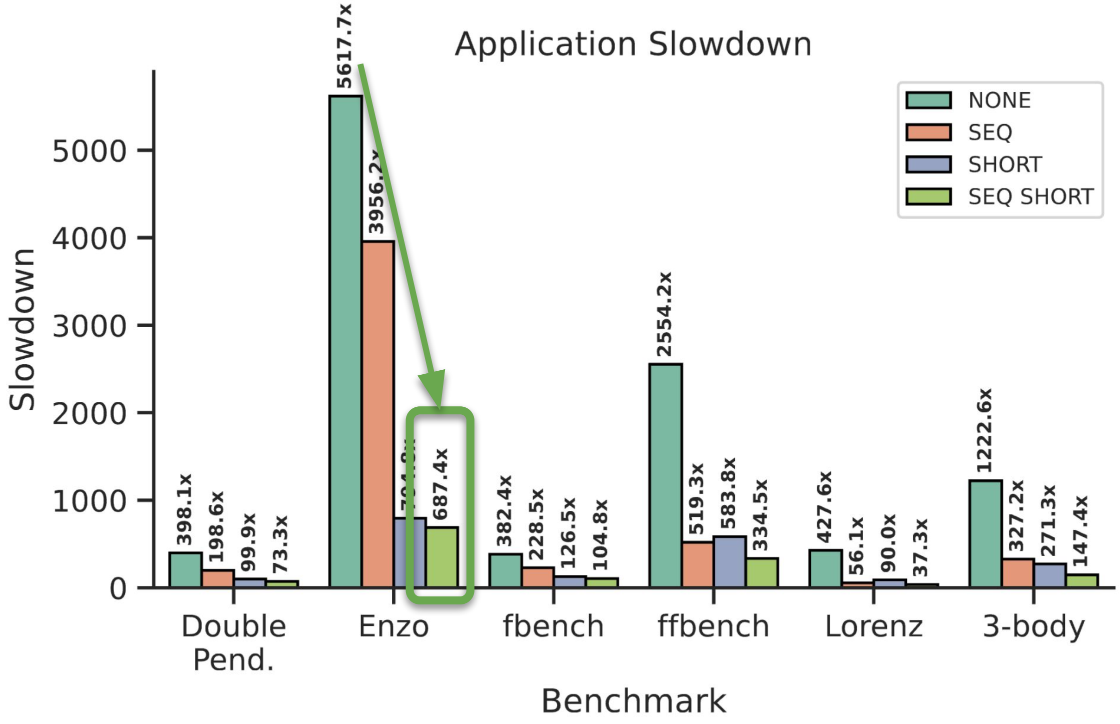


Results

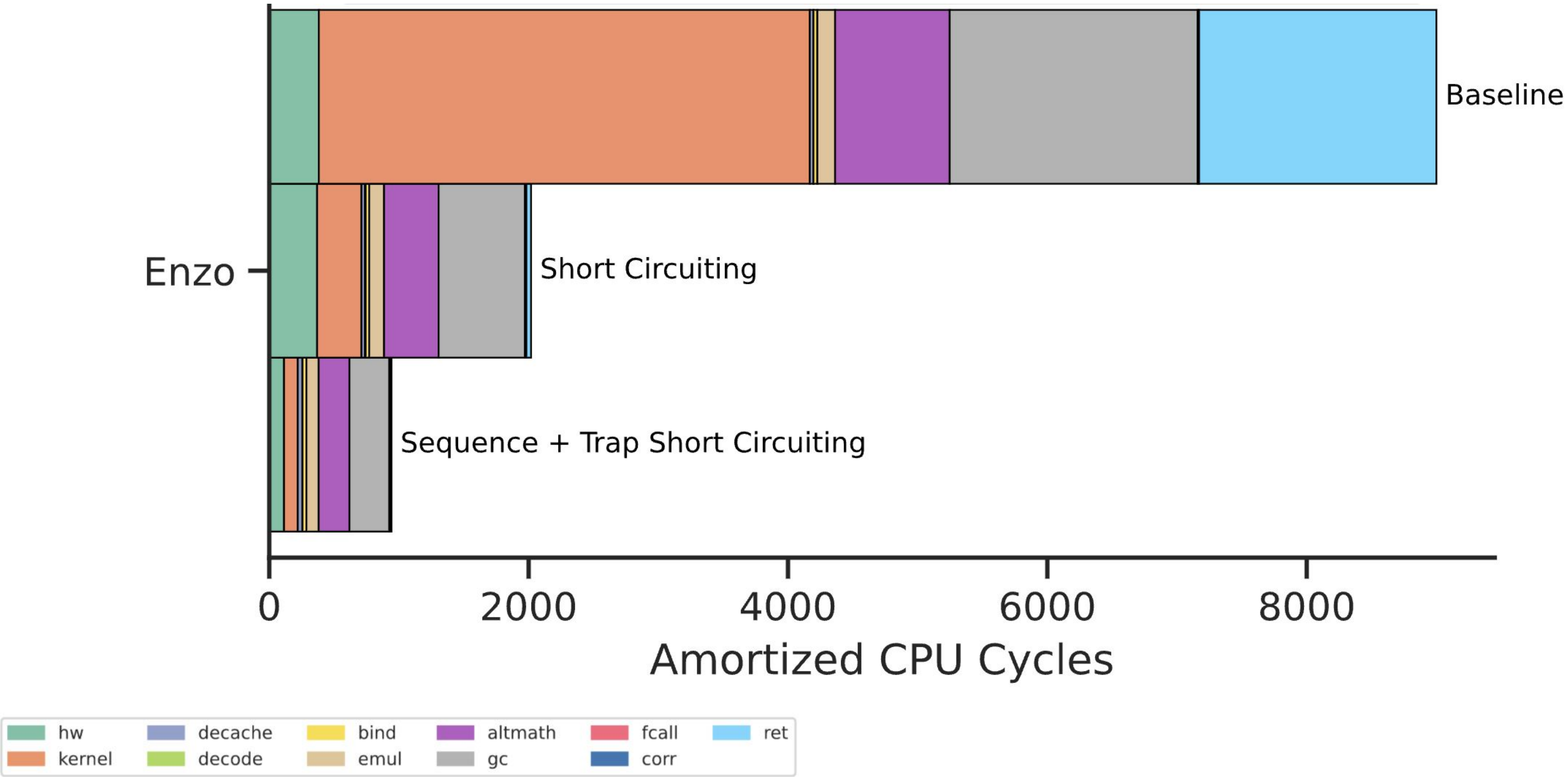
Altmath now dominates across the board



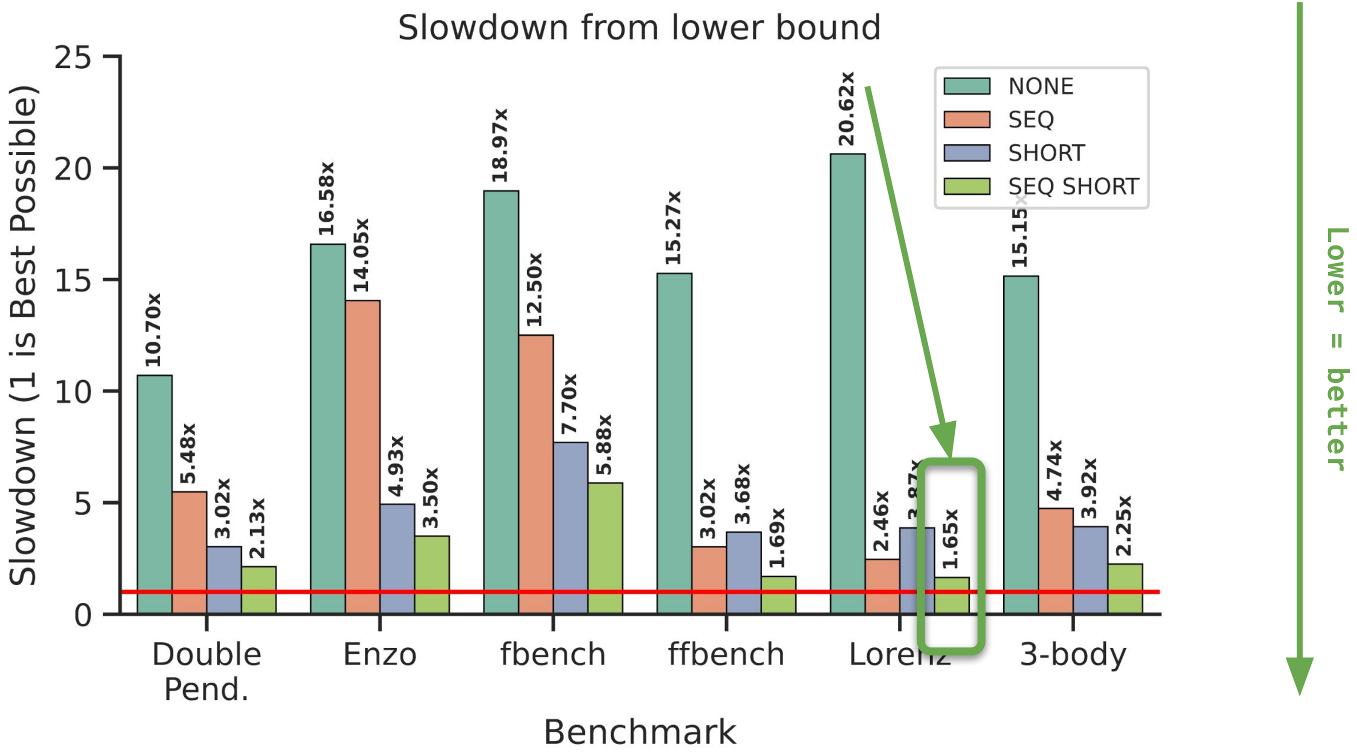
Using *boxed math*, overheads reduce by up to ~10x



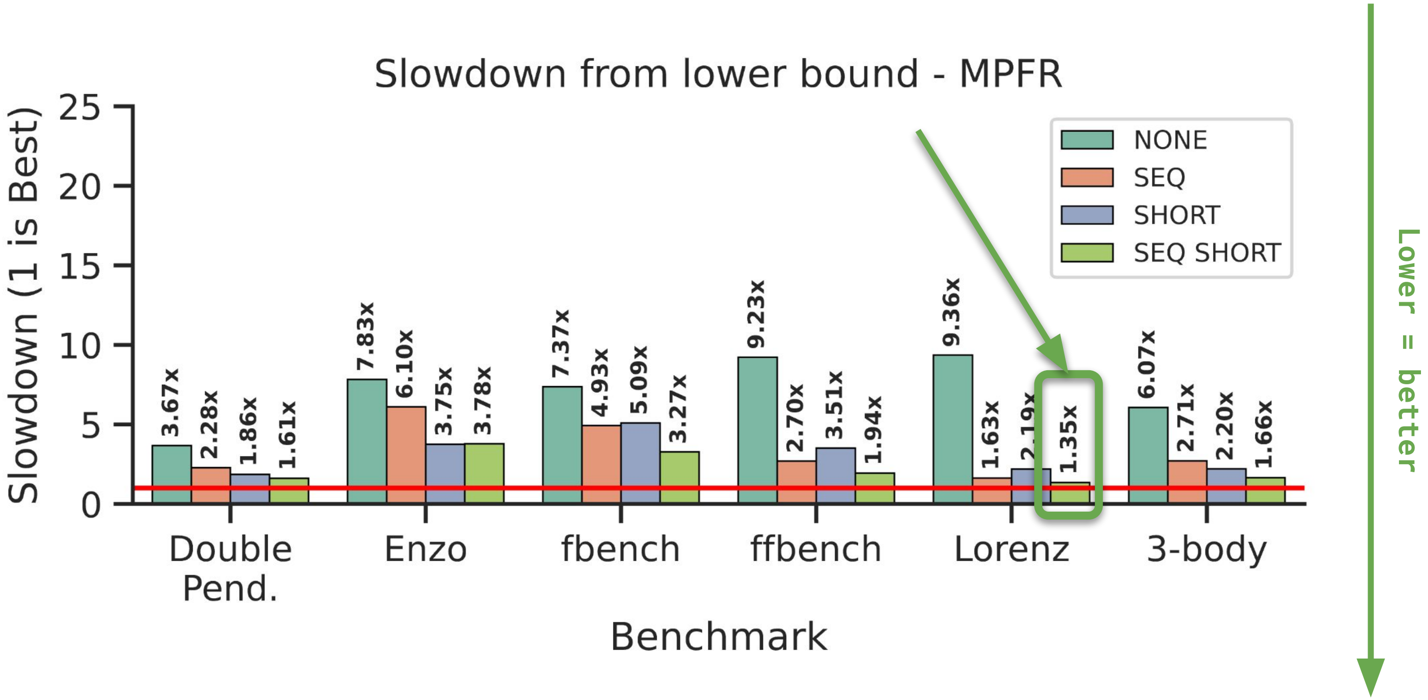
Virtualization overheads are also reduced



We are *much* closer to zero-cost virtualization



The overhead can get *even lower* with a more expensive altmath like MPFR



Conclusion

- We bypass signals with ***trap short circuiting***
- We emulate more instructions with ***sequence emulation***
- We reduce the time to do correctness analysis from **weeks** to **minutes**

- All of which reduces the overhead of virtualization ***around*** the **alternative math** library down to as low as **1.35x** with MPFR

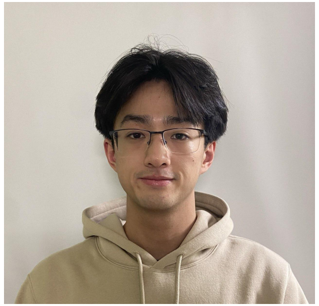
Sequence Emulation

Trap Short Circuiting

Profiler based correctness traps

Thanks!

Virtualization So Light, it *Floats*! Accelerating Floating Point Virtualization

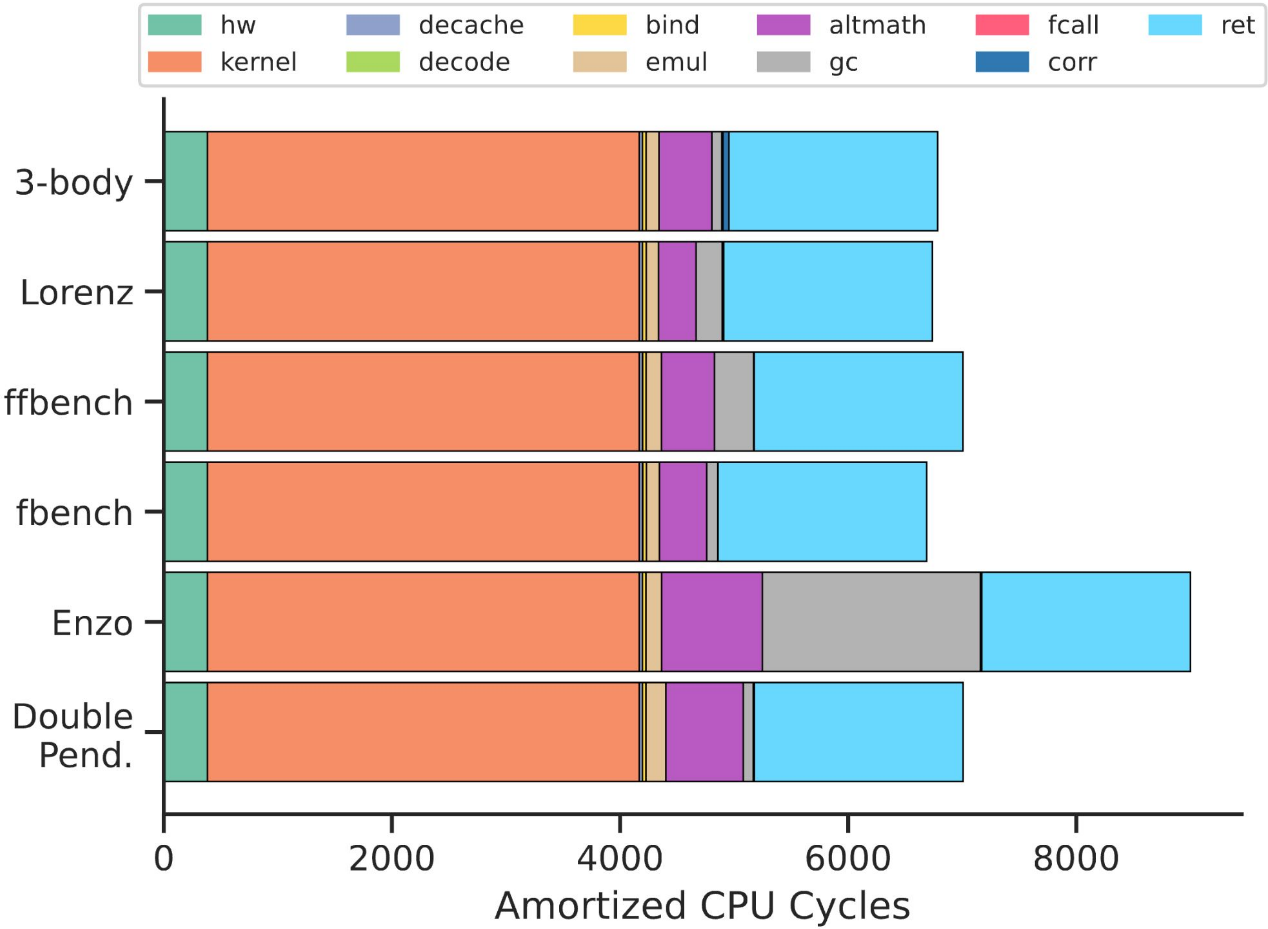


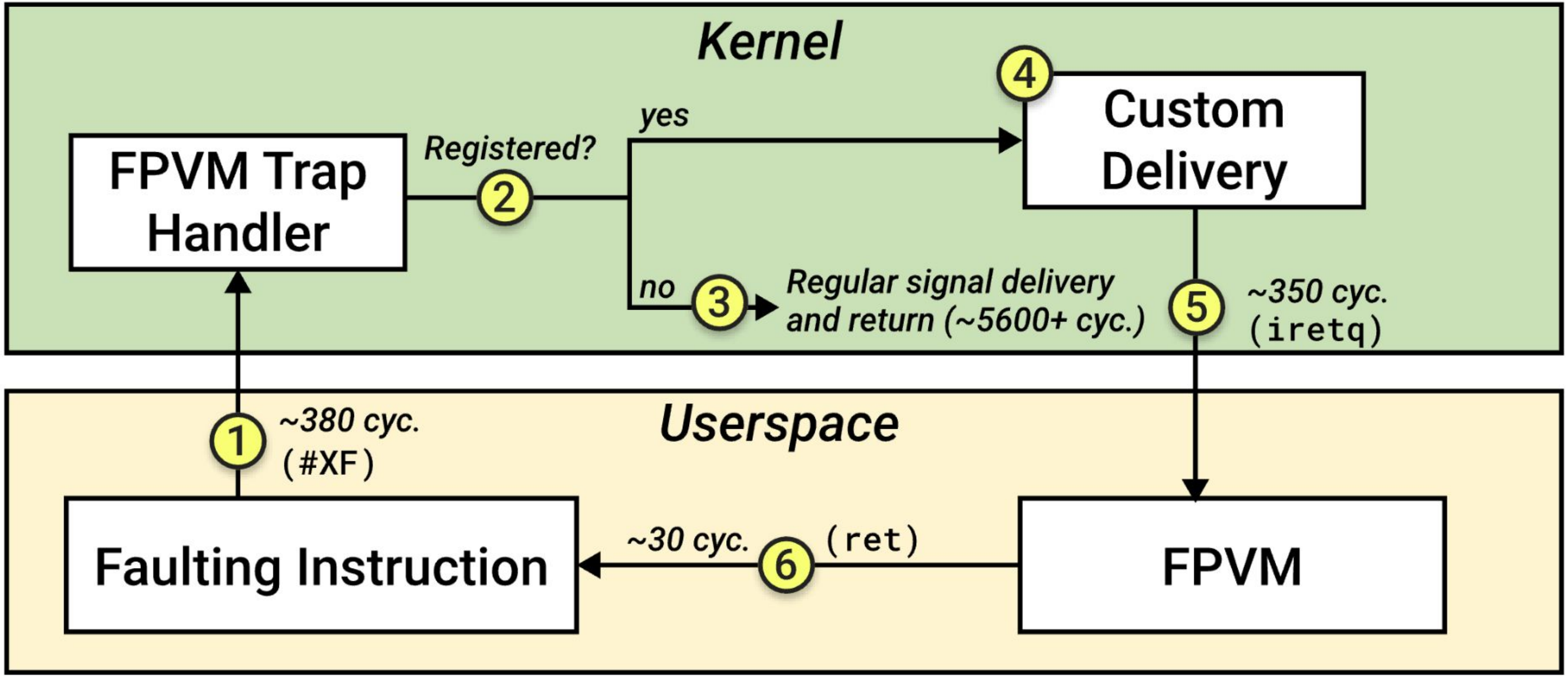
Nick Wanninger, Nadharm Dhiantravan, **Peter Dinda**

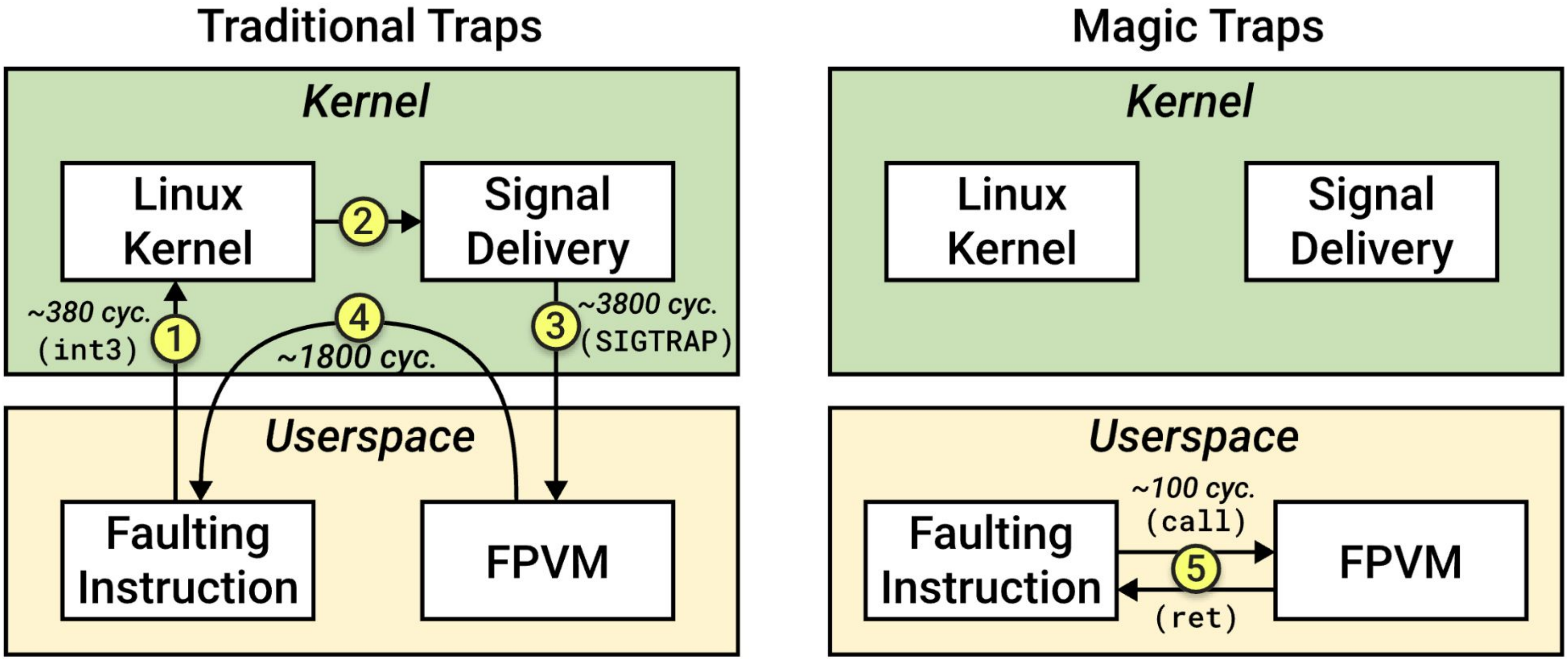
Download our paper!

Northwestern | Plab

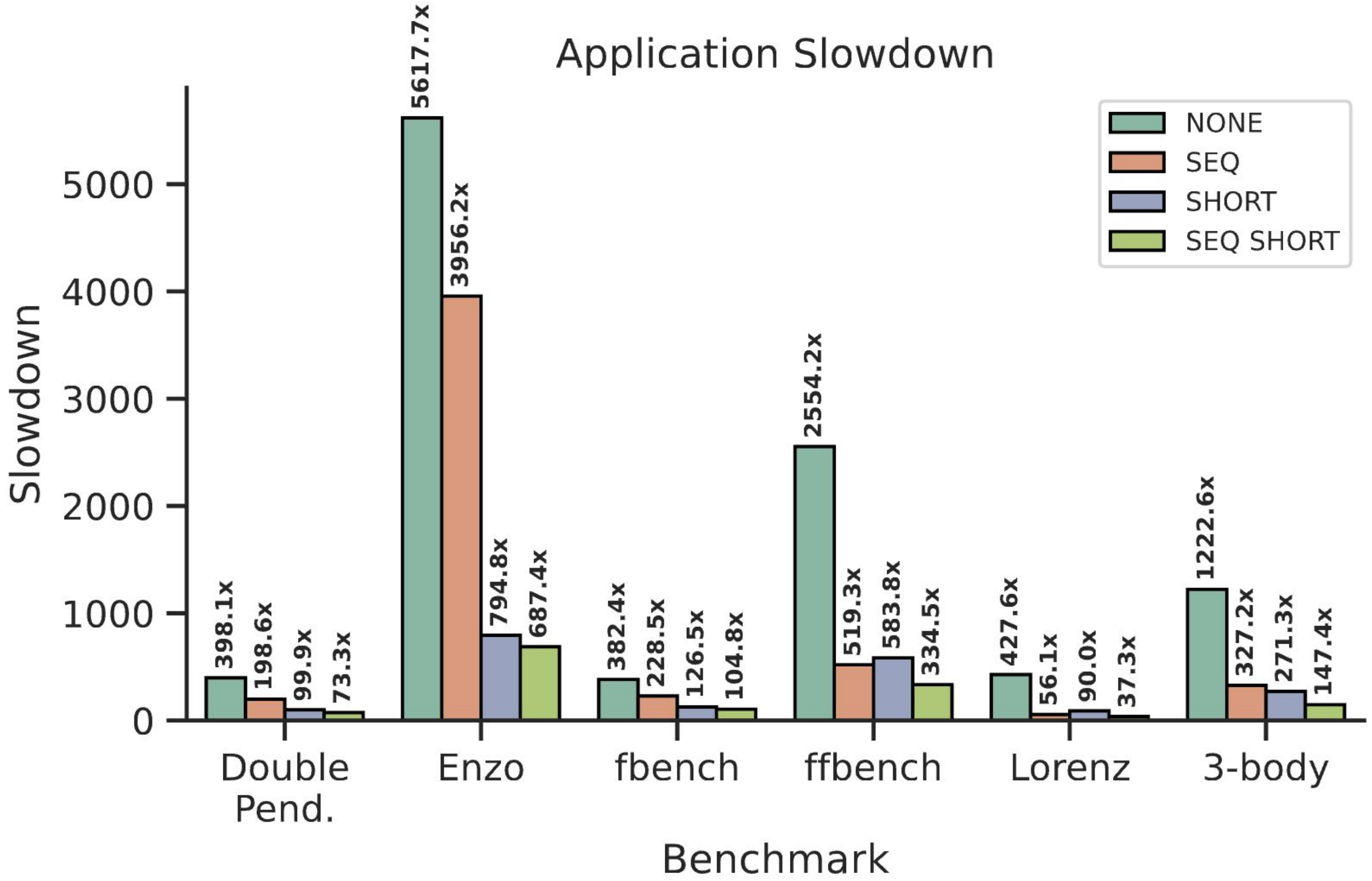
BACKUP SLIDES

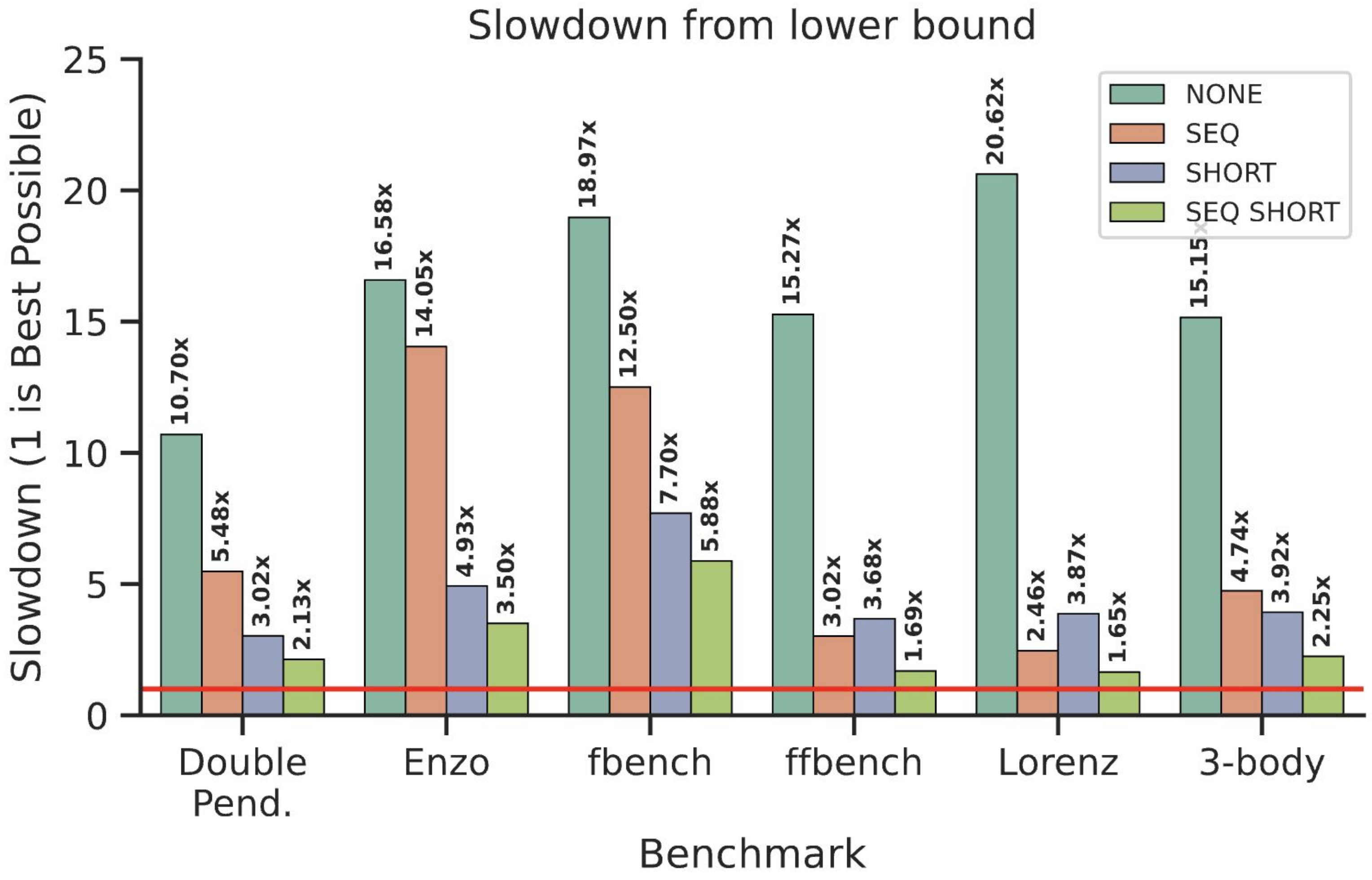

hw
kernel
decache
decode
bind
emul
altmath
gc
fcall
corr
ret
3-body
Lorenz
ffbench
fbench
Enzo
Double
Pend.
0
2000
4000
6000
8000
Amortized CPU Cycles

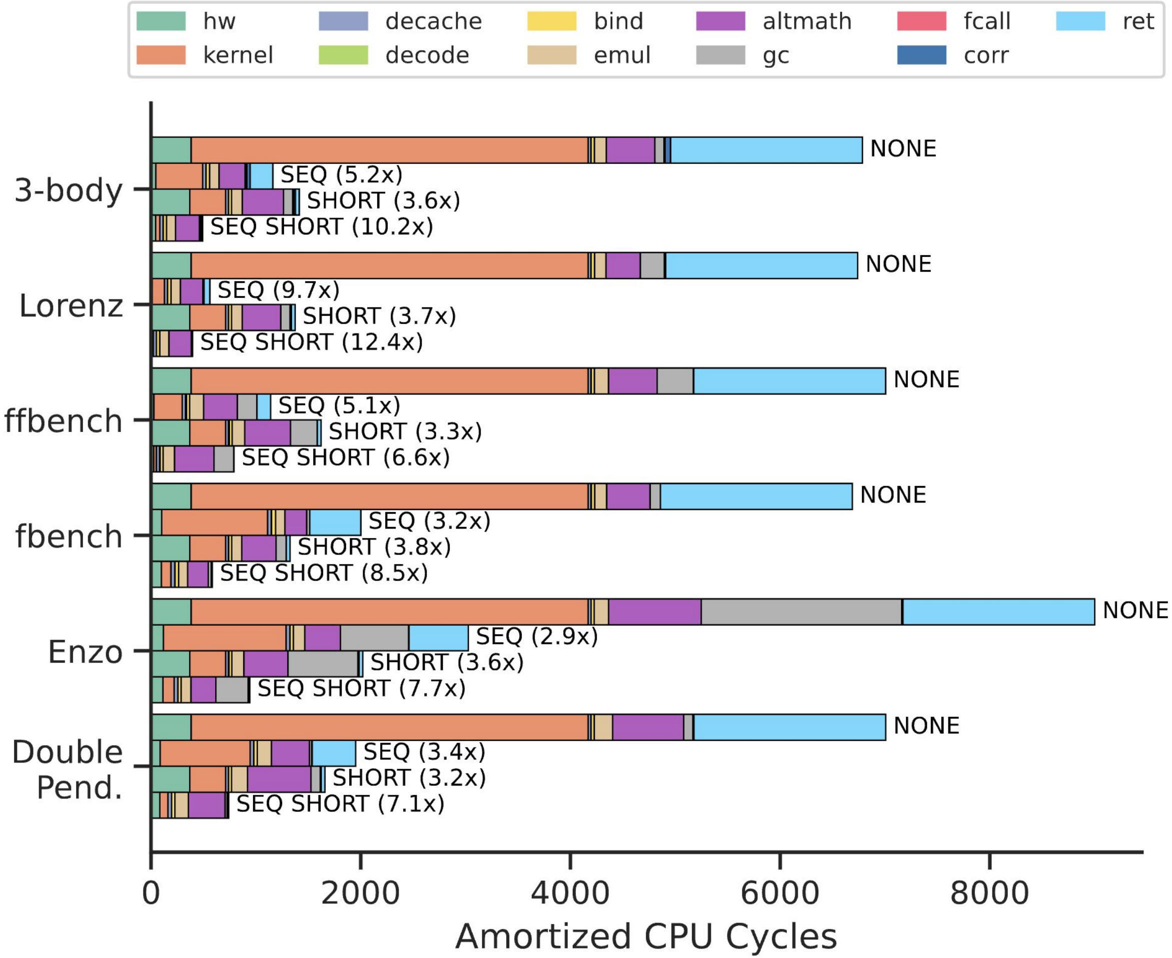

Kernel
4
Custom Delivery
yes
Registered?
FPVM Trap Handler
2
no
3
Regular signal delivery and return (~5600+ cyc.)
5
~350 cyc. (iretq)
Userspace
1
~380 cyc. (#XF)
Faulting Instruction
~30 cyc.
6
(ret)
FPVM



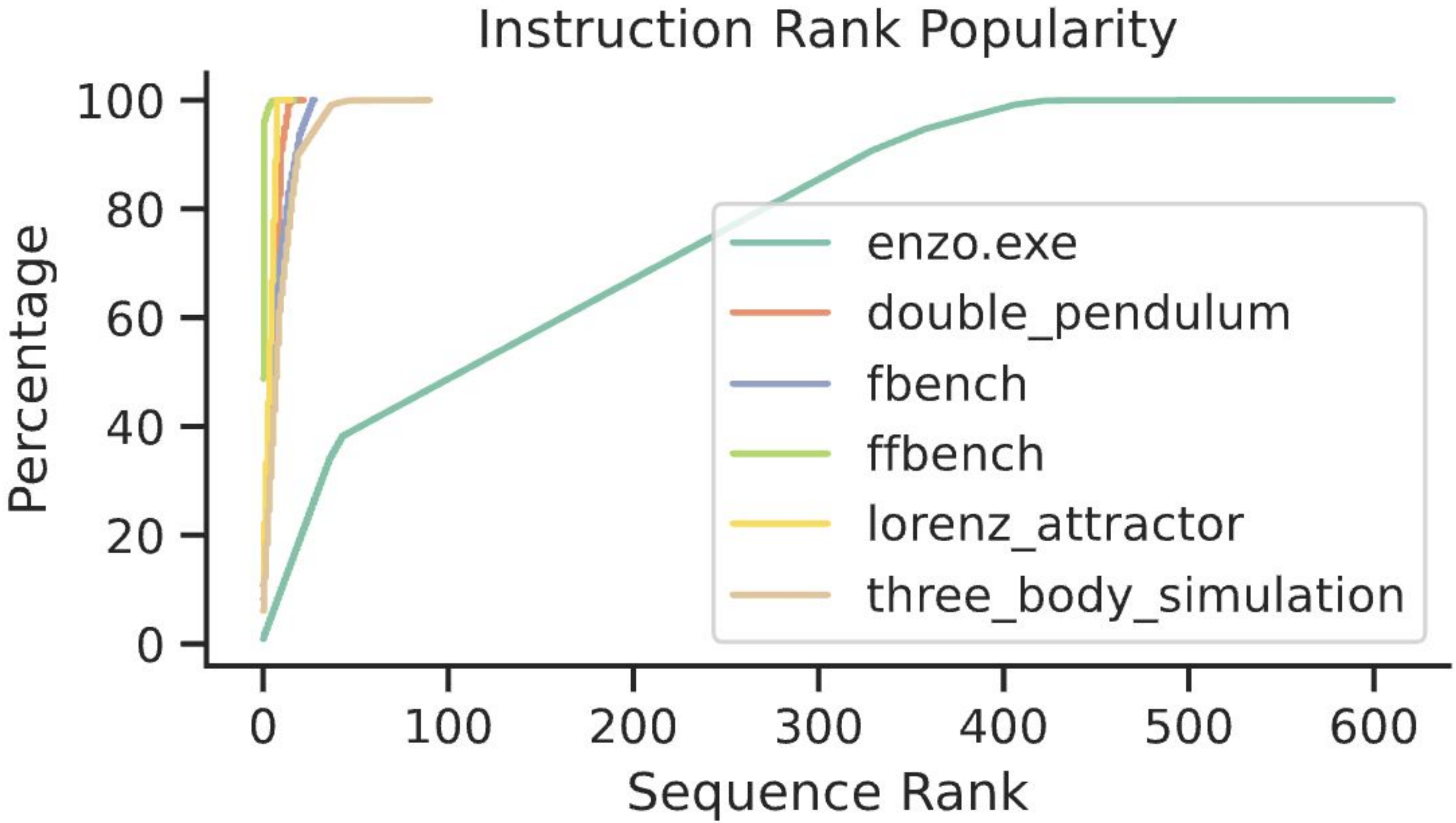
Magic Traps bypass the kernel


Application Slowdown
Slowdown
5000
4000
3000
2000
1000
0
NONE
SEQ
SHORT
SEQ SHORT
398.1x
198.6x
99.9x
73.3x
5617.7x
3956.2x
794.8x
687.4x
382.4x
228.5x
126.5x
104.8x
2554.2x
519.3x
583.8x
334.5x
427.6x
56.1x
90.0x
37.3x
1222.6x
327.2x
271.3x
147.4x
Double Pend.
Enzo
fbench
ffbench
Lorenz
3-body
Benchmark

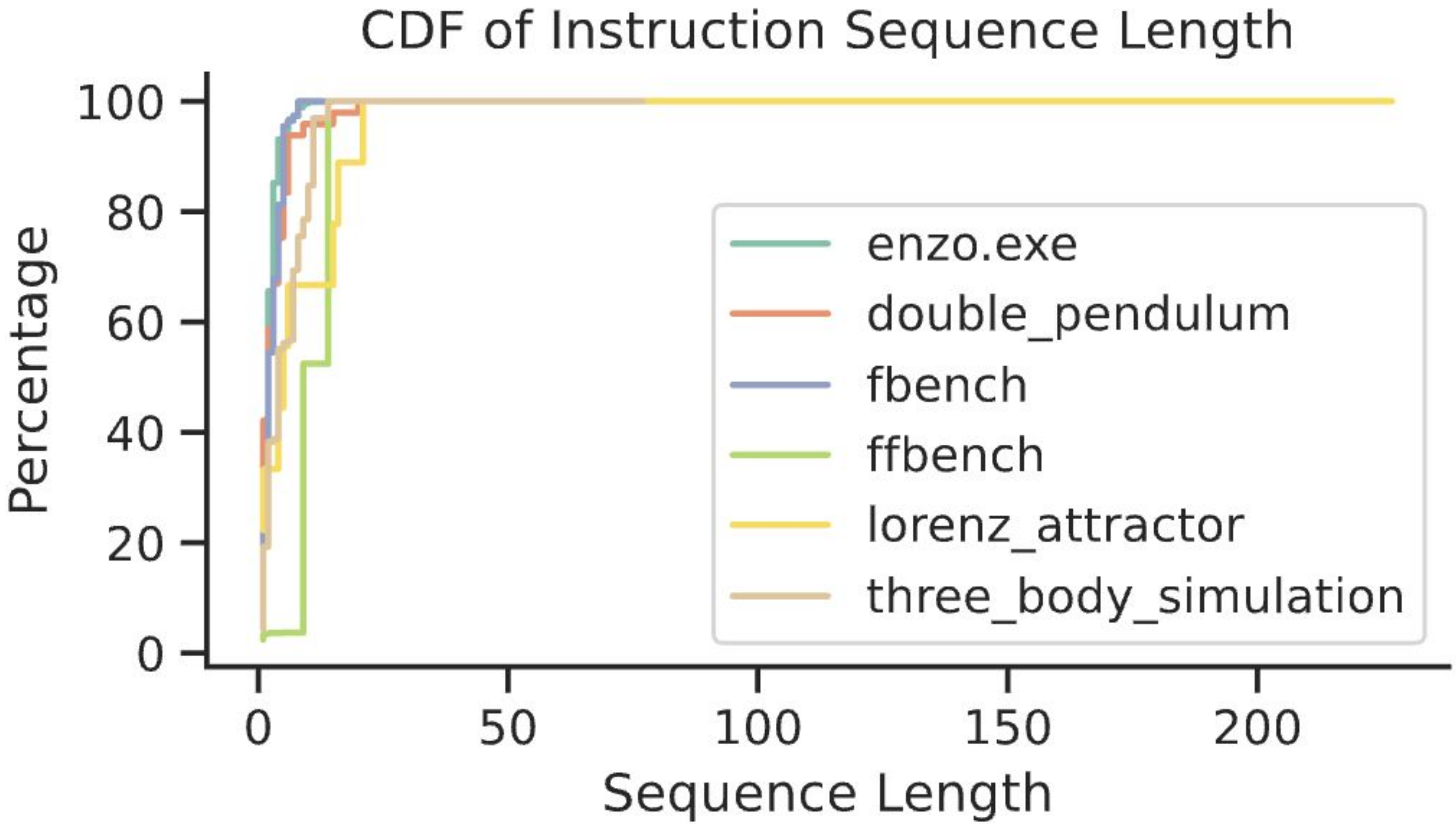

Slowdown from lower bound
Slowdown (1 is Best Possible)
25
20
15
10
5
0
NONE
SEQ
SHORT
SEQ SHORT
10.70x
5.48x
3.02x
2.13x
16.58x
14.05x
4.93x
3.50x
18.97x
12.50x
7.70x
5.88x
15.27x
3.02x
3.68x
1.69x
20.62x
2.46x
3.87x
1.65x
15.15x
4.74x
3.92x
2.25x
Double Pend.
Enzo
fbench
ffbench
Lorenz
3-body
Benchmark

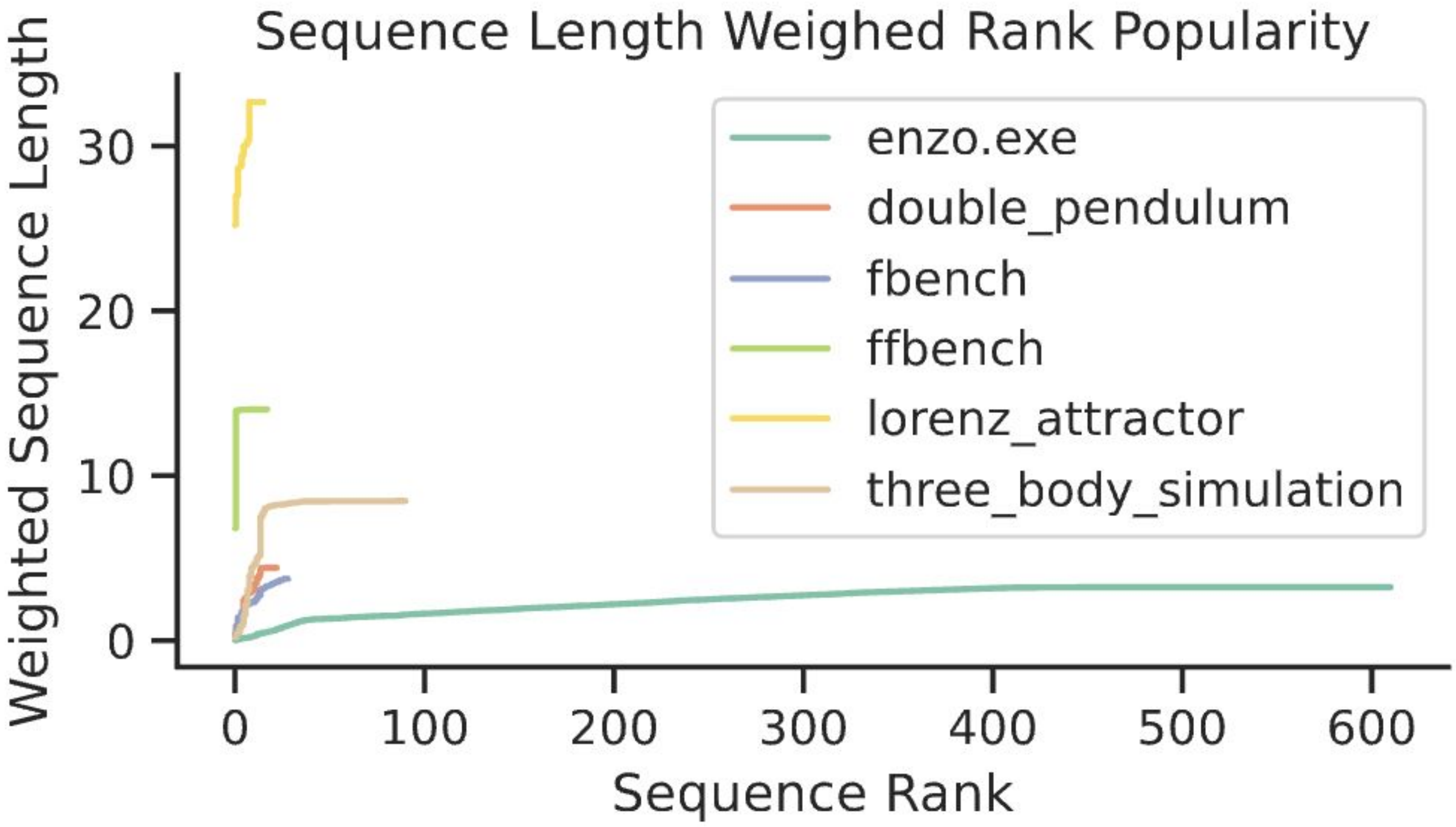


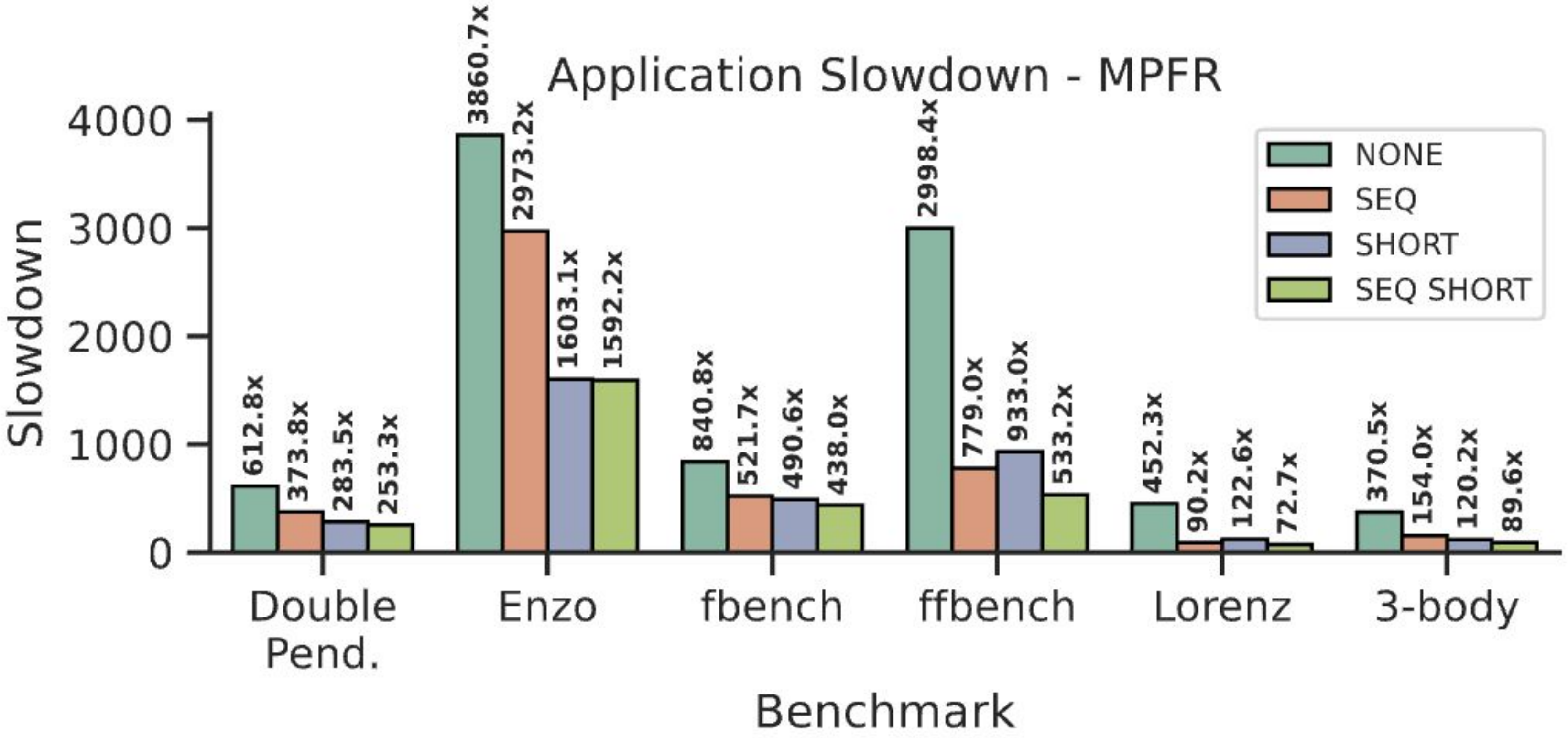
hw
kernel
decache
decode
bind
emul
altmath
gc
fcall
corr
ret
3-body
Lorenz
ffbench
fbench
Enzo
Double Pend.
NONE
SEQ (5.2x)
SHORT (3.6x)
SEQ SHORT (10.2x)
NONE
SEQ (9.7x)
SHORT (3.7x)
SEQ SHORT (12.4x)
NONE
SEQ (5.1x)
SHORT (3.3x)
SEQ SHORT (6.6x)
NONE
SEQ (3.2x)
SHORT (3.8x)
SEQ SHORT (8.5x)
NONE
SEQ (2.9x)
SHORT (3.6x)
SEQ SHORT (7.7x)
NONE
SEQ (3.4x)
SHORT (3.2x)
SEQ SHORT (7.1x)
0
2000
4000
6000
8000
Amortized CPU Cycles

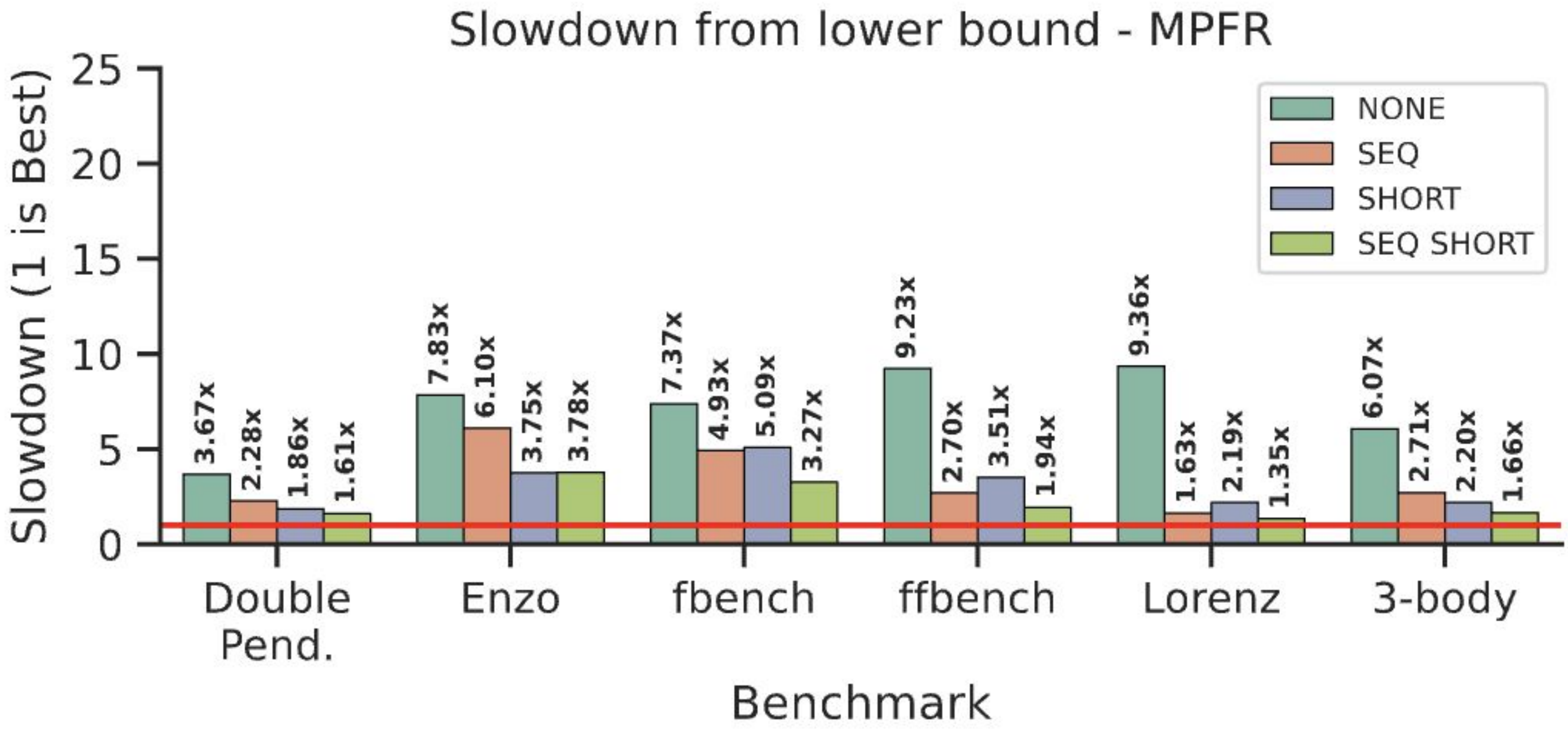


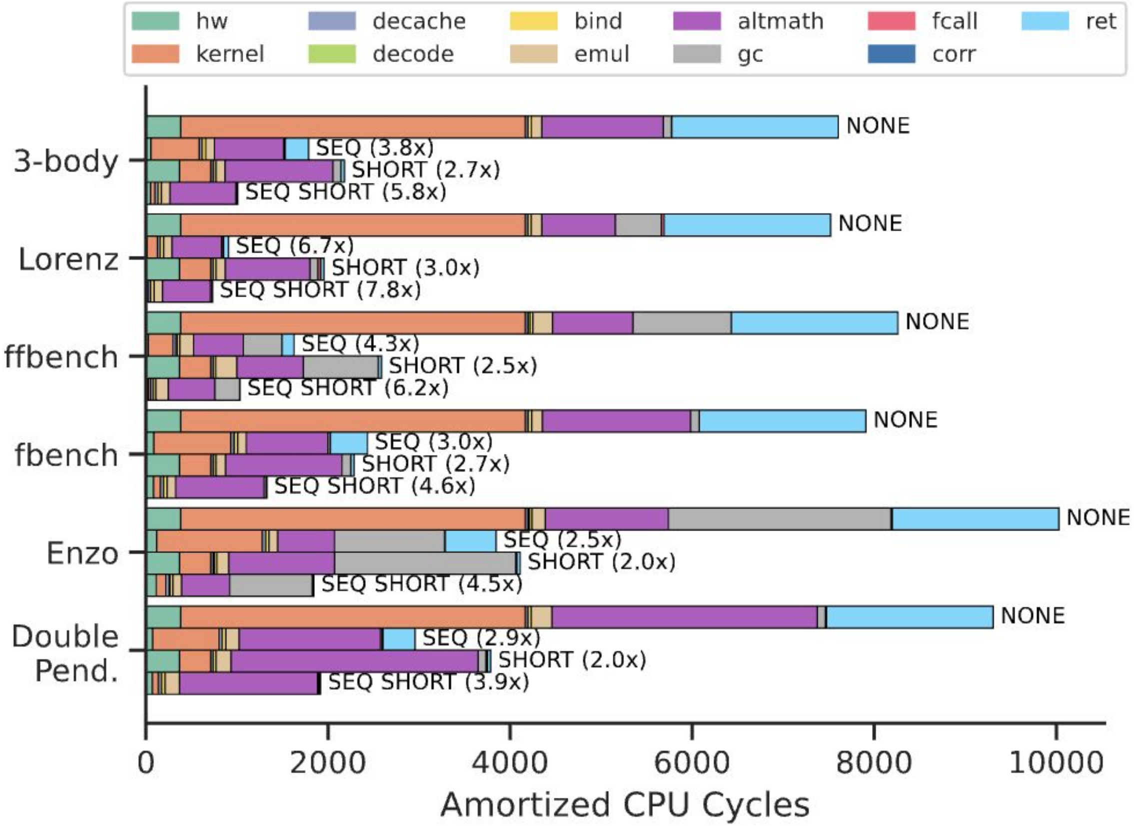
Instruction Rank Popularity
Percentage
100
80
60
40
20
0
0
100
200
300
400
500
600
Sequence Rank
enzo.exe
double_pendulum
fbench
ffbench
lorenz_attractor
three_body_simulation


CDF of Instruction Sequence Length
Percentage
Sequence Length
0
20
40
60
80
100
0
50
100
150
200
enzo.exe
double_pendulum
fbench
ffbench
lorenz_attractor
three_body_simulation


Sequence Length Weighed Rank Popularity
Weighted Sequence Length
30
20
10
0
0
100
200
300
400
500
600
Sequence Rank
enzo.exe
double_pendulum
fbench
ffbench
lorenz_attractor
three_body_simulation


Application Slowdown - MPFR
Slowdown
4000
3000
2000
1000
0
3860.7x
2973.2x
1603.1x
1592.2x
612.8x
373.8x
283.5x
253.3x
840.8x
521.7x
490.6x
438.0x
2998.4x
779.0x
933.0x
533.2x
452.3x
90.2x
122.6x
72.7x
370.5x
154.0x
120.2x
89.6x
NONE
SEQ
SHORT
SEQ SHORT
Double Pend.
Enzo
fbench
ffbench
Lorenz
3-body
Benchmark


Slowdown from lower bound - MPFR
Slowdown (1 is Best)
25
20
15
10
5
0
NONE
SEQ
SHORT
SEQ SHORT
3.67x
2.28x
1.86x
1.61x
7.83x
6.10x
3.75x
3.78x
7.37x
4.93x
5.09x
3.27x
9.23x
2.70x
3.51x
1.94x
9.36x
1.63x
2.19x
1.35x
6.07x
2.71x
2.20x
1.66x
Double Pend.
Enzo
fbench
ffbench
Lorenz
3-body
Benchmark



MPFR altmath overheads