# Fractured Processes

## Adaptive, Fine-Grained Process Abstractions

Thanumalayan Sankaranarayana Pillai,

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

# Process

Central abstraction in modern operating systems

- Provides a virtual machine to users

Many tools, techniques built around processes

- Environments and wrappers (Valgrind, memory limits …)
- Automatic restarts for fault tolerance
- Replication for N-versioning

# Monolithic Applications

Modern applications are mostly monolithic

- Use a single (or few) big process
- Processes used more as threads, not as co-operating tasks
- Example: Microsoft Office, Apple iPhoto, PostgreSQL

Impractical for process-based tools, techniques

# Big Processes: Disadvantage

## Automatic restarts

- GUI applications: User-visible restarts
- Restart time too high

## Environments such as Valgrind

- Too expensive to apply to entire process

## N-version techniques

- Too expensive to apply

# Previous Solutions

## Use small processes

- High, persistent overhead
- Cost of a function call: a few cycles
- Cost of two context switches: 4 to 6 µs

## Process-like abstractions

- Micro-reboots[OSDI '04], Band-aid patching[HotDep '07]
- EJB, COM, OSGi
- Not as general as processes, requires re-inventing tools

# Fractured Processes

Small processes, RPC-like interaction

But, isolate only necessary parts as processes

- Set of processes changes at each run
- Use most optimal "boundary" for isolation

Target existing C applications

- Including multi-threaded applications
- Allow incremental conversion (*vs* rewriting in another language)

Supporting paradigms such as restarting

# Outline

Introduction

Example: Pidgin

Fracture Usage & Features

- Isolation
- Restarts, Replication, and Sampling

Programming Annotations

Implementation

Evaluation

# Outline

- Introduction
- Example: Pidgin
- Fracture Usage & Features
  - Isolation
  - Restarts, Replication, and Sampling
- Programming Annotations
- Implementation
- Evaluation

# Example

Pidgin: Real-world instant messenger

- GUI application
- Event-based
- Extensible library of IM protocols and plugins

Four (previously patched) bugs in Pidgin

- Two memory leaks
- Two buffer overflows

# Pidgin: Memory Leaks

Valgrind tool: Usual strategy to test memory leaks

- Runtime tool, requires running an application workload

Pidgin's leaks occur only in rare workloads

- Workloads cannot be tested during development

Applying Valgrind has intolerable GUI lag

- Cannot ask users to run Valgrind in deployment

# Fractured Processes Solution

"Crowd-sourcing Valgrind"

1. In each deployment, isolate a small part of Pidgin
   - Two processes: small isolated part, rest of Pidgin
2. Run Valgrind on top of the isolated part
   - Overhead tolerable in each user's deployment

Overhead measured: 20% worst-case

# Pidgin: Buffer Overflows

The culprit buffer overflows …

- Occur for a certain class of inputs
- Cause repeated crashes in some deployments

Possible tolerance strategy: Rx [SOSP '05]

- Try restarting with different process environments until application can run
- Leaks can be tolerated using an LD_PRELOAD trick

However, Rx-restarts cause GUI interruptions

# Fractured Processes Solution

“Micro-Rx Tolerance”

After first crash, isolate GUI into separate process

Try different environments on the non-GUI part

Extended solution: Search for problematic part

- Divide non-GUI part further, performing a binary search
- Isolate problematic part in an optimal manner from GUI

# Outline

- Introduction
- Example: Pidgin
- Fracture Usage & Features
  - Isolation
  - Restarts, Replication, and Sampling
- Programming Annotations
- Implementation
- Evaluation

# Isolation

1. Programer divides program into *modules*

# Isolation

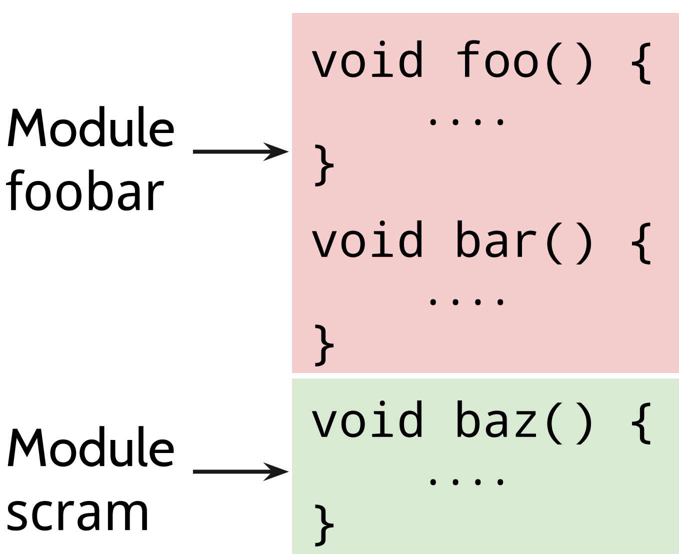
1. Programer divides program into *modules*

```
void foo() {
    ....
}
void bar() {
    ....
}
void baz() {
    ....
}
```

Example program

# Isolation

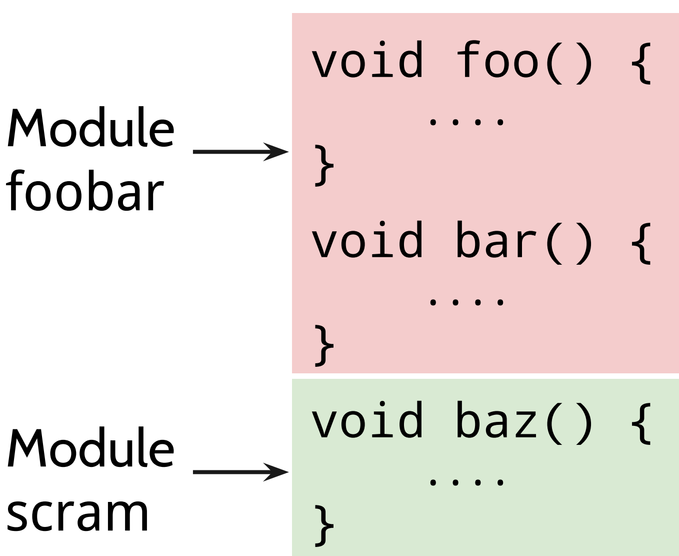
1. Programer divides program into *modules*



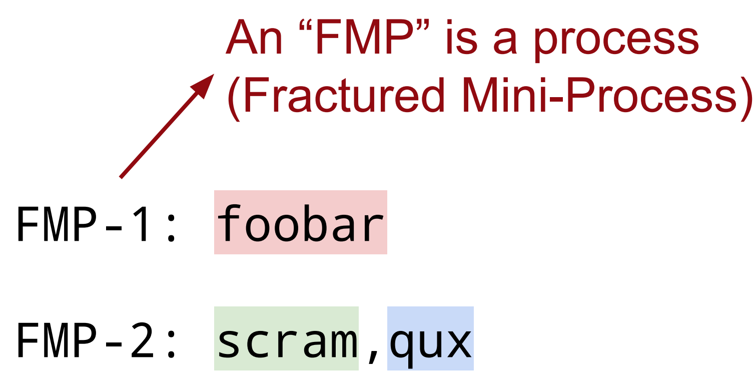

Example program

# Isolation

2. At runtime, modules are composed into processes
   - Specified by an user or administrator
   - Can vary with each run!




Example program




Example runtime configuration

# Isolation

3. Each FMP can have an *environment* (FMP-E)
   - Examples: Valgrind, LD_PRELOAD, memory limits ....

Module foobar →

```
void foo() {
    ....
}
void bar() {
    ....
}
```

Module scram →

```
void baz() {
    ....
}
```

Example program

```
FMP-1: foobar with valgrind

FMP-2: scram,qux
```

Example runtime configuration

# Isolation: Runtime Configurations

1. All modules in a single FMP
   - For normal usage: High performance, no isolation

2. One FMP per module
   - Usually not used: Low performance, high isolation

# Intelligent Partitioning

3. Suspicious module in a separate FMP
   - Used when needed, fair performance

4. Intelligent Partitioning
   - Isolating a module as a separate FMP will have overhead
   - Optimization: Move coupled modules also to isolated FMP
   - Fracture can predict most optimal boundary for isolating a given module

# Outline

Introduction

Example: Pidgin

Fracture Usage & Features

- Isolation
- Restarts, Replication, and Sampling

Programming Annotations

Implementation

Evaluation

# Restarts, Replication, Sampling

1. Programmer labels each module with *capabilites*
   - Three capabilities: Restartable, replicable, samplable

Module foobar (restartable, replicable) →

```
void foo() {
    ....
}
void bar() {
    ....
}
```

Module scram (replicable) →

```
void baz() {
    ....
}
```

Example program

```
FMP-1: foobar
FMP-2: scram,qux
```

Example runtime configuration

# Restarts, Replication, Sampling

2. Capable FMPs can be configured to restart etc.

Module foobar (restartable, replicable) →

```
void foo() {
    ....
}
void bar() {
    ....
}
```

Module scram (replicable) →

```
void baz() {
    ....
}
```

Example program

```
FMP-1: foobar
```
(restart on crash)

```
FMP-2: scram,qux
```
(replicate 2 times)

Example runtime configuration

# Restarts

On a crash, FMP is restarted with crashed request

- Restarts etc. are transparent to other FMPs!

Use-case: Simple restarts for fault tolerance

- Faster than full restarts
- Restarting *foo* module does not affect *GUI* module

FMP-E can be changed during each restart

- Rx-like fault tolerance[SOSP '05], Software Rejuvenation[FTCS '95]

# Replication

N processes are run for the same FMP

- Requests are supplied to all processes, and responses collated
- If responses are different, appropriate action taken

Use-case: N-version fault detection

- Low overhead than full N-version
- Full N-versioning might not be possible (Eg: with GUI)

FMP-E can be different for different versions

# Sampling

N processes are run for the same FMP, but ...

- Requests are distributed among the different processes

FMP-E can be different for different versions

Use-case: Applying Valgrind-like tool to one version

- Further reduces tool overhead: only few requests are slowed

# Outline

Introduction

Example: Pidgin

Fracture Usage & Features

- Isolation
- Restarts, Replication, and Sampling

<u>Programming Annotations</u>

Implementation

Evaluation

# Identifying Modules

Any set of functions can be a module ...

With some restrictions

- Global variables: Not allowed across modules
- OS resources (like files): Not shared between modules
- Pointer parameters, heap allocations: Special semantics

# Pointer Parameters, Heap Allocations

Pointer parameters are assigned a special semantic

- *On-demand copy-by-value-result*
- Data referred to is copied to callee, then copied back
- Might require additional annotation in few cases
- Affects synchronization in multi-threaded code

Fracture allows data to *belong* to only one module

- Pointers referring to data are in the same module as data

# Labeling Module Capabilities

1. Imagine modules as micro-servers

2. Modules will possess all three capabilities if they
   a. Deterministically interact with other modules in each thread,
   b. Are idempotent, and
   c. Possess a few more conditions
   - If modules are not idempotent and deterministic, more complex (less restrictive) conditions can be used

# Outline

- Introduction
- Example: Pidgin
- Fracture Usage & Features
  - Isolation
  - Restarts, Replication, and Sampling
- Programming Annotations
- Implementation
- Evaluation

# Implementation

## FMP interaction: Shared memory queues

## Restarts, replication, sampling

- Restarts: Queue entries logged in memory and replayed
- Replication: Entries mirrored and collated
- Sampling: Entries multiplexed and de-multiplexed

## Intelligent partitioning help

- Module interaction recorded with a training workload
- Mincut (graph) algorithm finds optimal isolation boundary

# Outline

Introduction

Example: Pidgin

Fracture Usage & Features

- Isolation
- Restarts, Replication, and Sampling

Programming Annotations

Implementation

Evaluation

# Evaluation

## Four real-world applications

- Null-httpd: Multi-threaded, CGI-capable web server ()
- NTFS-3g: Single threaded FUSE file system (30K LOC)
- SSHFS: Multi-threaded FUSE file system, in-memory cache
- Pidgin

## Different strategies for dividing into modules

- Null-httpd: Each function made into separate module
- SSHFS: Divided into logical parts of code

# Programming Overhead

57 pointer parameters required annotation

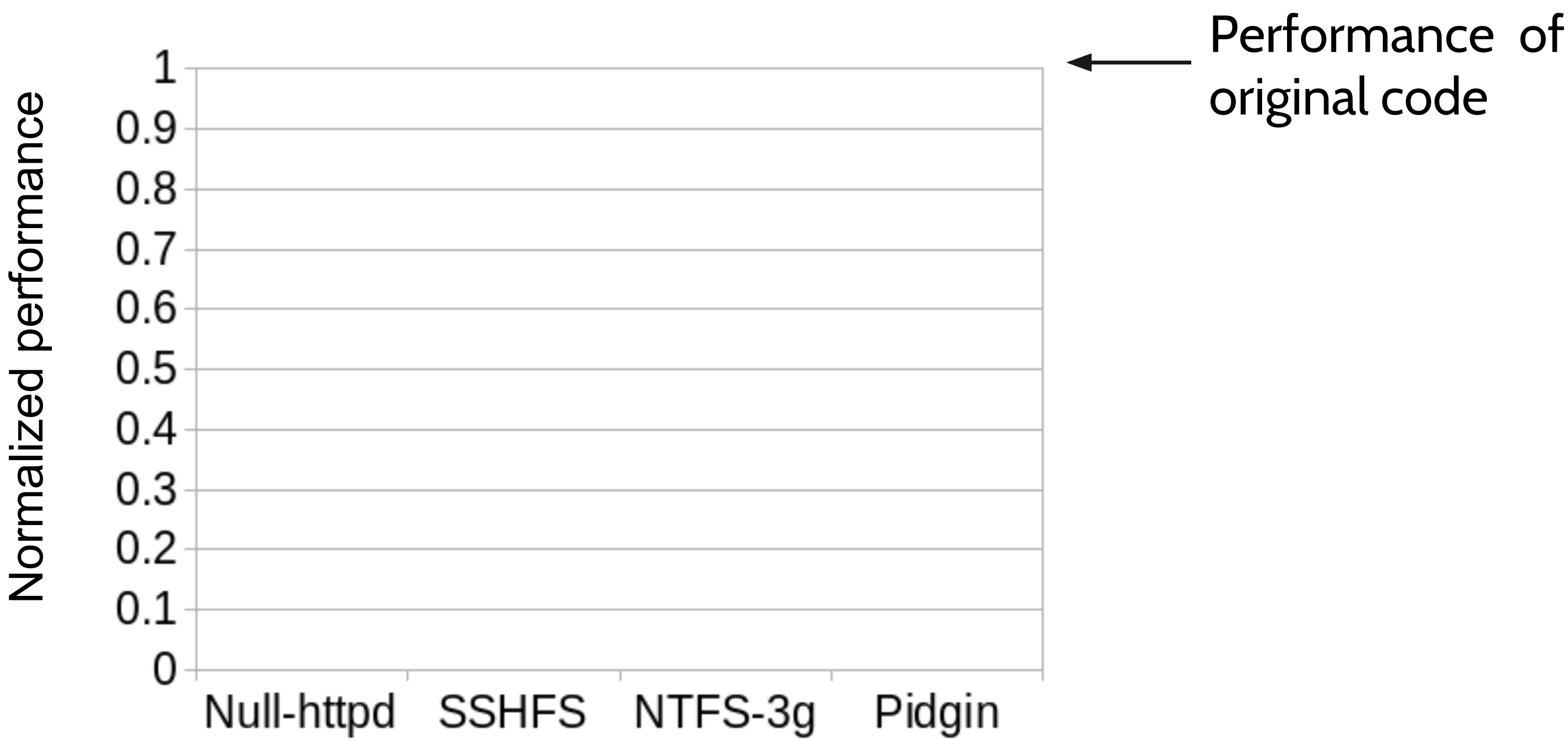
- Context: 104 modules

Hard: Identifying ownership of heap allocations

Effort easy for logical divisions already in code

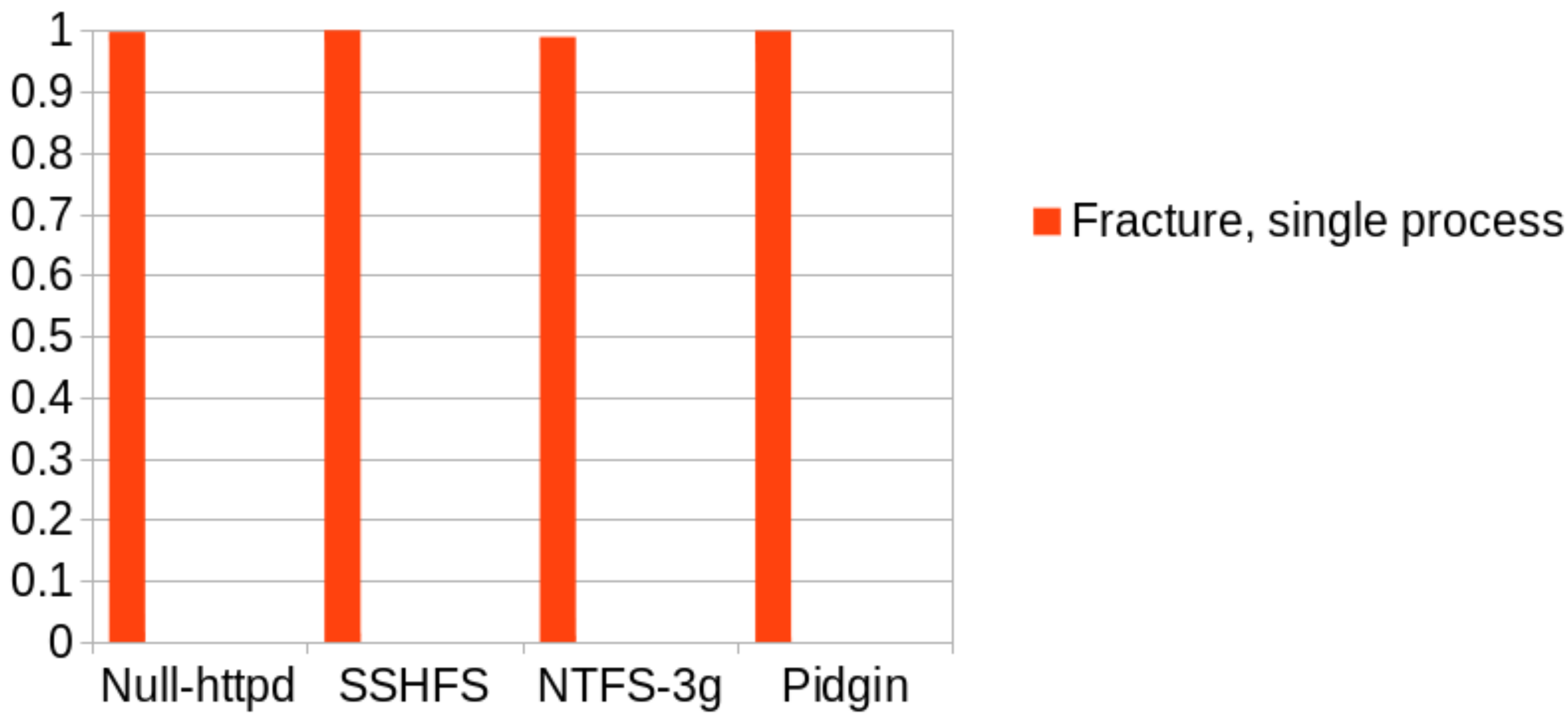
- Harder for goals such as “put every function into a module”

# Isolation Performance



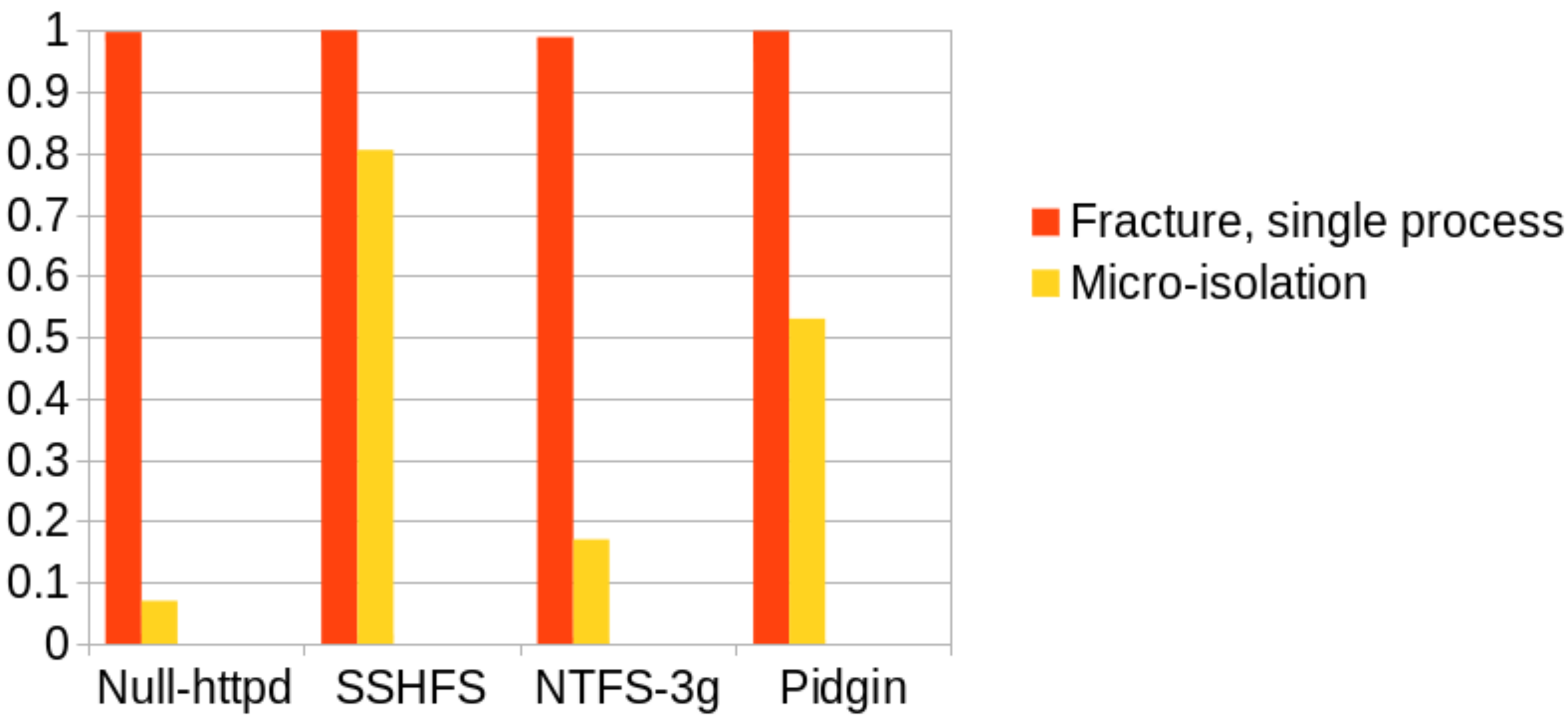
# No Isolation

Fracture with all modules in a single FMP: Insignificant overhead



# One process for each module

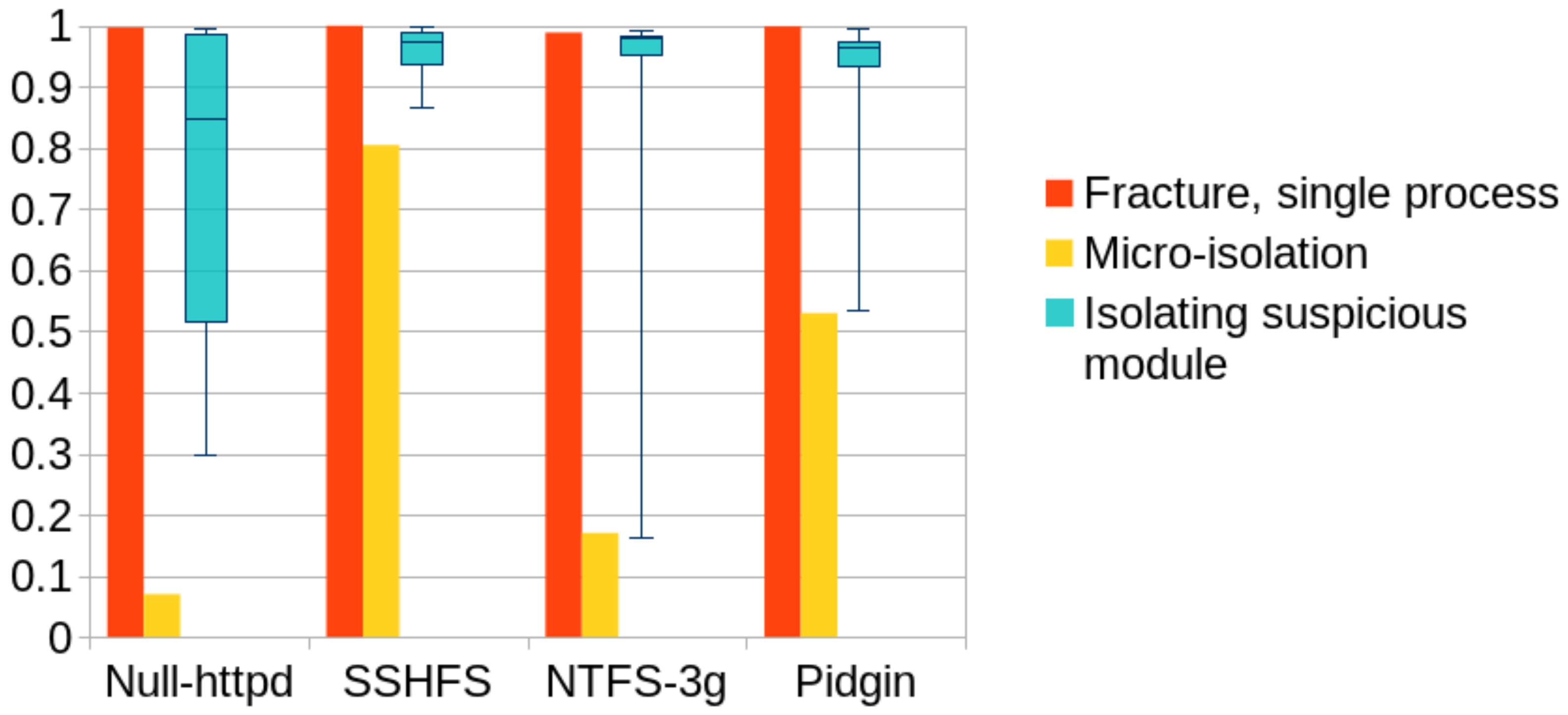
Fracture with each module in separate FMP: High overhead



# Isolating Individual Modules

Many modules can be isolated with tolerable overhead

Some staggering modules



# Restart, Replication, Sampling Performance

Restarts: Same overhead as isolation

- In our experiments, restarting took less than 1 ms

Replication: N-times overhead of isolation

Sampling: Same overhead of isolation

- Additionally, overhead of FMP-E applied

# Conclusion

An ecosystem is built around processes

Applications forgo the ecosystem for performance

On-demand RPC-style interaction, with some additional thought, can provide best of both worlds

# Thank you!

Questions?