The Case for the $\vec{V}$ector $\vec{O}$perating $\vec{S}$ystem

Vijay Vasudevan,
David G. Andersen, Michael Kaminsky

Carnegie Mellon University and Intel Labs

A webserver

Req1

```
accept(...)
stat(...)
open(f1)
fcntl(...)
fcntl(...)
...
```

Req2

```
accept(...)
stat(f2)
open(f2)
fcntl(...)
fcntl(...)
...
```

A webserver

Req1

```
accept(...)
stat(...)
open(f1)
fcntl(...)
fcntl(...)
...
```

Req2

```
accept(...)
stat(f2)
open(f2)
fcntl(...)
fcntl(...)
...
```

A scalable, parallel webserver

Req1

```
accept(...)
stat(...)
open(f1)
fcntl(...)
fcntl(...)
...
```

Req2

```
accept(...)
stat(f2)
open(f2)
fcntl(...)
fcntl(...)
...
```

Req3

```
accept(...)
stat(f3)
open(f3)
fcntl(...)
fcntl(...)
...
```

A scalable, parallel webserver

```
accept(...)        accept(...)        accept(...)
stat(f1)           stat(f2)           stat(f3)
open(f1)           open(f2)           open(f3)
```

Req1 **Req2** **Req3**

```
...                ...                ...
```

A scalable, parallel webserver

`accept(...)`	`accept(...)`	`accept(...)`
`stat(f1)`	`stat(f2)`	`stat(f3)`
`open(f1)`	`open(f2)`	`open(f3)`
Req1	**Req2**	**Req3**
...	...	...

A scalable, parallel webserver



Req1	Req2	Req3
accept(...)	accept(...)	accept(...)
stat(f1)	stat(f2)	stat(f3)
open(f1)	open(f2)	open(f3)
...	...	...

A scalable, parallel webserver



```
stat(f1)
open(f1)
```

```
vec_accept(...)
stat(f2)
open(f2)
```

```
stat(f3)
open(f3)
```

A scalable, parallel webserver



```
                 vec_accept(...)
                 vec_stat([f1,f2,f3])
open(f1)            open(f2)            open(f3)
```

A scalable, parallel webserver

```
                vec_accept(...)
                vec_stat([f1,f2,f3])
open(f1) {            open(f2)             open(f3)
  context switch
  alloc()
  copy(f1)
  path_resolve(f1)
    acl_check(f1)
    h = hash(f1)
    lookup(h)
  read(f1)
  dealloc()
  context switch
}
```

A scalable, parallel webserver



```
vec_accept(...)
vec_stat([f1,f2,f3])
```

```
open(f1) {
  context switch
  alloc()
  copy(f1)
  path_resolve(f1)
    acl_check(f1)
    h = hash(f1)
    lookup(h)
  read(f1)
  dealloc()
  context switch
}
```

```
open(f2) {
  context switch
  alloc()
  copy(f2)
  path_resolve(f2)
    acl_check(f2)
    h = hash(f2)
    lookup(h)
  read(f2)
  dealloc()
  context switch
}
```

```
open(f3) {
  context switch
  alloc()
  copy(f3)
  path_resolve(f3)
    acl_check(f3)
    h = hash(f3)
    lookup(h)
  read(f3)
  dealloc()
  context switch
}
```

A scalable, parallel webserver

```
vec_accept(...)
vec_stat([f1,f2,f3])
vec_open([f1,f2,f3]) {
      context switch
      vec_alloc()
      vec_copy([f1,f2,f3])
      vec_path_resolve([f1,f2,f3])
         acl_check([f1,f2,f3])
         h = hash([f1,f2,f3])
         lookup(h)
      vec_read([f1,f2,f3])
      dealloc()
      context switch
}
```

A vectored webserver

```
vec_accept(...)
vec_stat([f1,f2,f3])
vec_open([f1,f2,f3]) {
        context switch
        vec_alloc()
        vec_copy([f1,f2,f3])
        vec_path_resolve([f1,f2,f3])
           acl_check([f1,f2,f3])
           h = hash([f1,f2,f3])
           lookup(h)
        vec_read([f1,f2,f3])
        dealloc()
        context switch
}
```

A vectored webserver



```
vec_accept(...)
vec_stat([f1,f2,f3])
vec_open([f1,f2,f3]) {
        context switch
        vec_alloc()
        vec_copy([f1,f2,f3])
        vec_path_resolve([f1,f2,f3])
           acl_check([f1,f2,f3])
           h = hash([f1,f2,f3])
           lookup(h)
        vec_read([f1,f2,f3])
        dealloc()
        context switch
}
```

Eliminate N-1 context switches

A vectored webserver



```
vec_accept(...)
vec_stat([f1,f2,f3])
vec_open([f1,f2,f3]) {
        context switch
        vec_alloc()
        vec_copy([f1,f2,f3])
        vec_path_resolve([f1,f2,f3])
           acl_check([f1,f2,f3])
           h = hash([f1,f2,f3])
           lookup(h)
        vec_read([f1,f2,f3])
        dealloc()
        context switch
}
```

Reduce path resolutions

A vectored webserver



```
vec_accept(...)
vec_stat([f1,f2,f3])
vec_open([f1,f2,f3]) {
        context switch
        vec_alloc()
        vec_copy([f1,f2,f3])
        vec_path_resolve([f1,f2,f3])
            acl_check([f1,f2,f3])
            h = hash([f1,f2,f3])
            lookup(h)
        vec_read([f1,f2,f3])
        dealloc()
        context switch
}
```

Use SSE to hash filenames

A vectored webserver



```
vec_accept(...)
vec_stat([f1,f2,f3])
vec_open([f1,f2,f3]) {
        context switch
        vec_alloc()
        vec_copy([f1,f2,f3])
        vec_path_resolve([f1,f2,f3])
          acl_check([f1,f2,f3])
          h = hash([f1,f2,f3])
          lookup(h)
        vec_read([f1,f2,f3])
        dealloc()
        context switch
}
```

Search dentry list once

VOS core ideas

Known: Batching syscalls improves throughput

- Amortizes per-execution cost
- Applies regardless of similarity of batched work

“SIMD” vectorization improves efficiency

- Eliminates redundant instructions in || execution
- Frees up resources to allow more work to be done
- Enables algorithmic optimizations

VOS core ideas

Known: Batching syscalls improves throughput

- Amortizes per-execution cost
- Applies regardless of similarity of batched work

“SIMD” vectorization improves efficiency

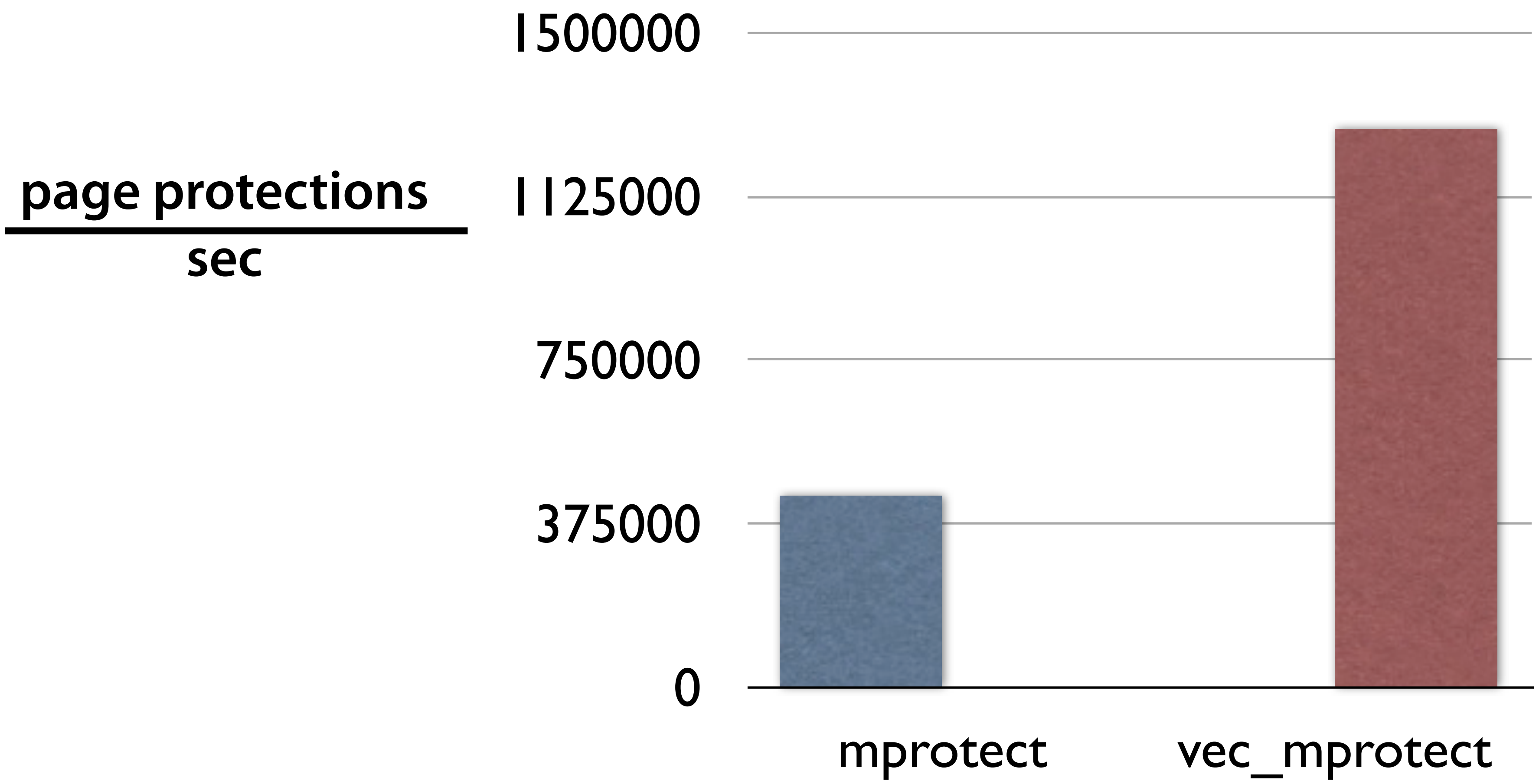
- Eliminates redundant instructions in || execution
- Frees up resources to allow more work to be done
- Enables algorithmic optimizations

One concrete example: mprotect

One difficult challenge: managing divergence

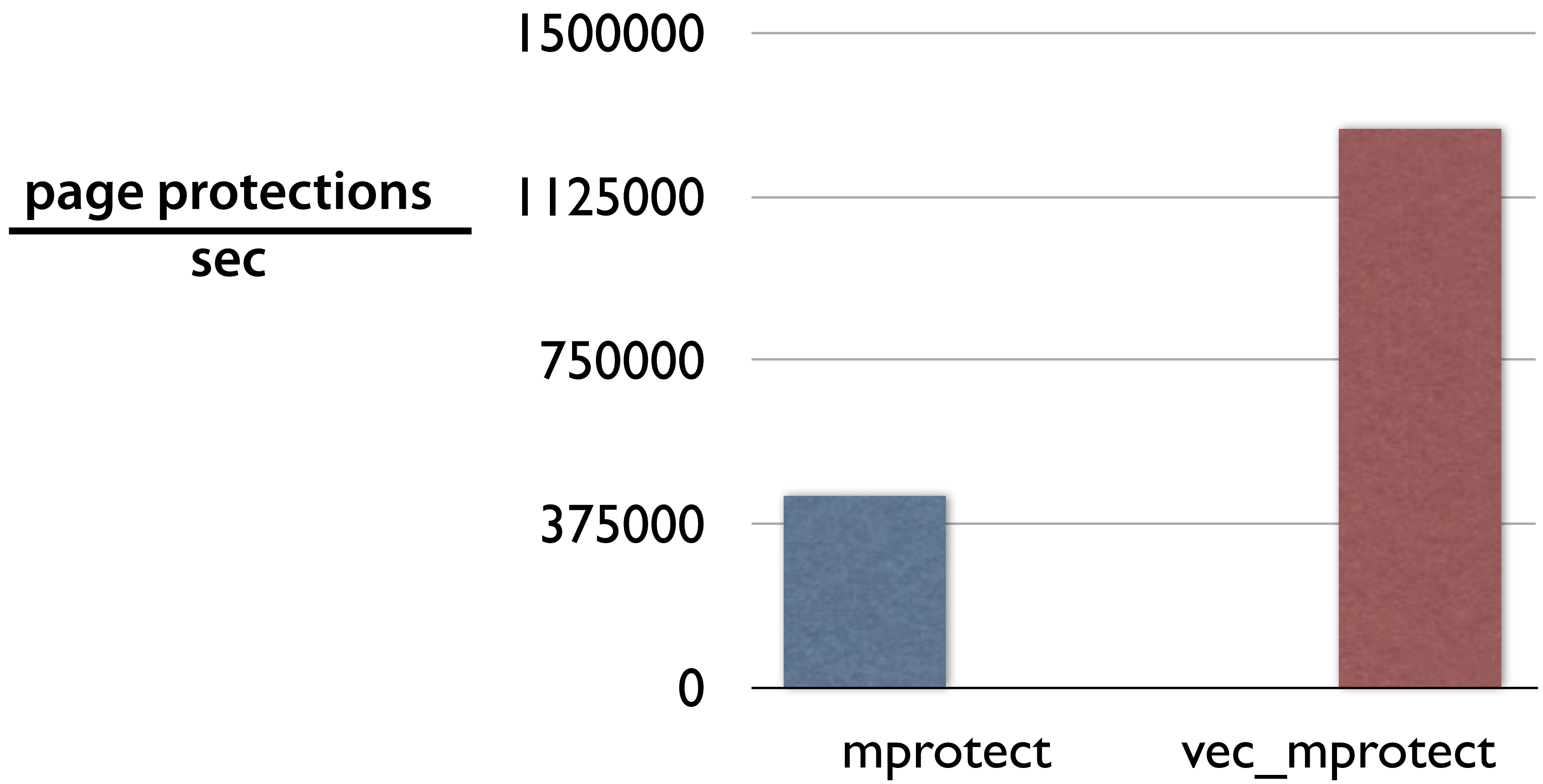
One possible implementation path

Speeding up memory protection



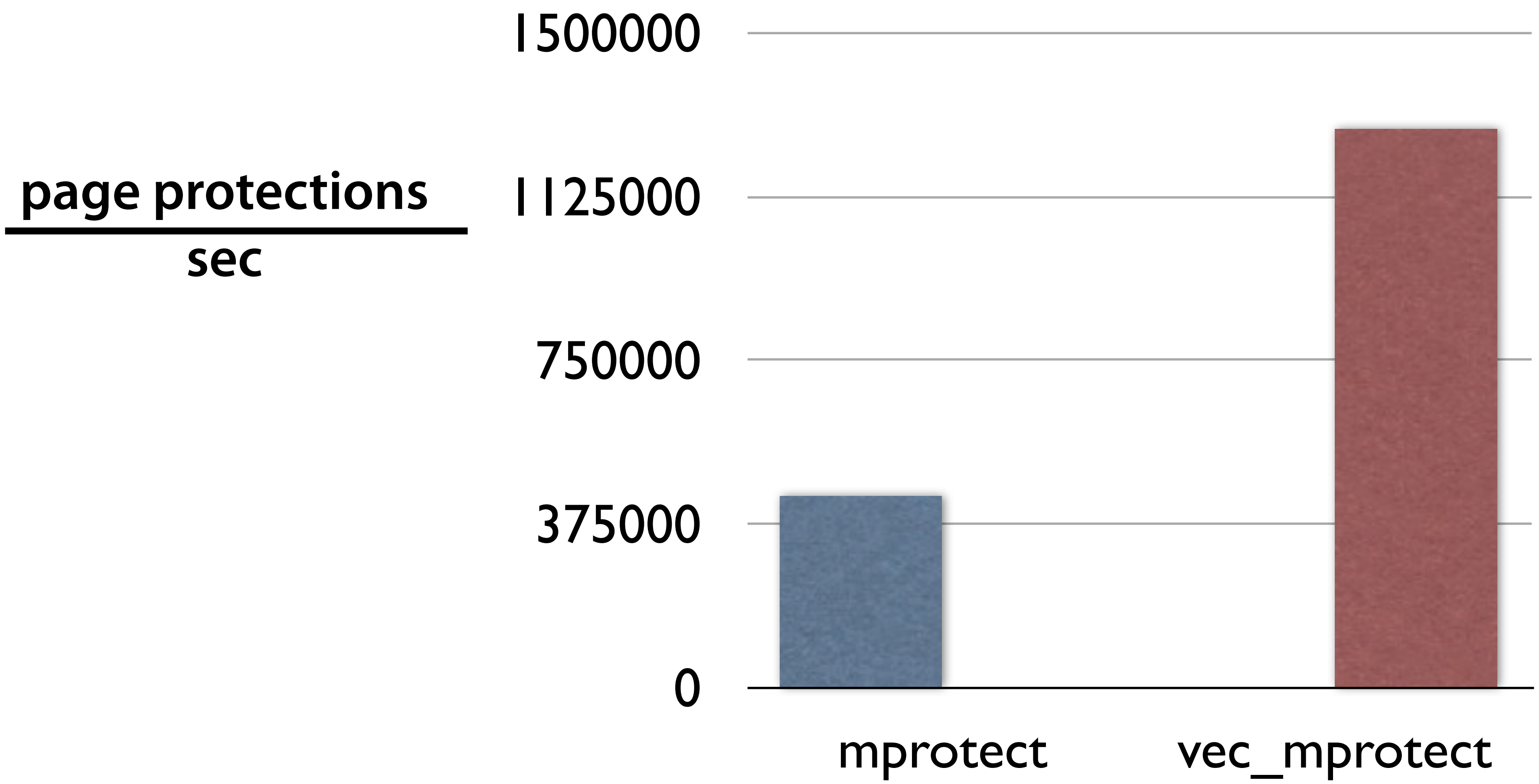
Data courtesy of Iulian Moraru

Speeding up memory protection



vec_mprotect techniques:

Speeding up memory protection

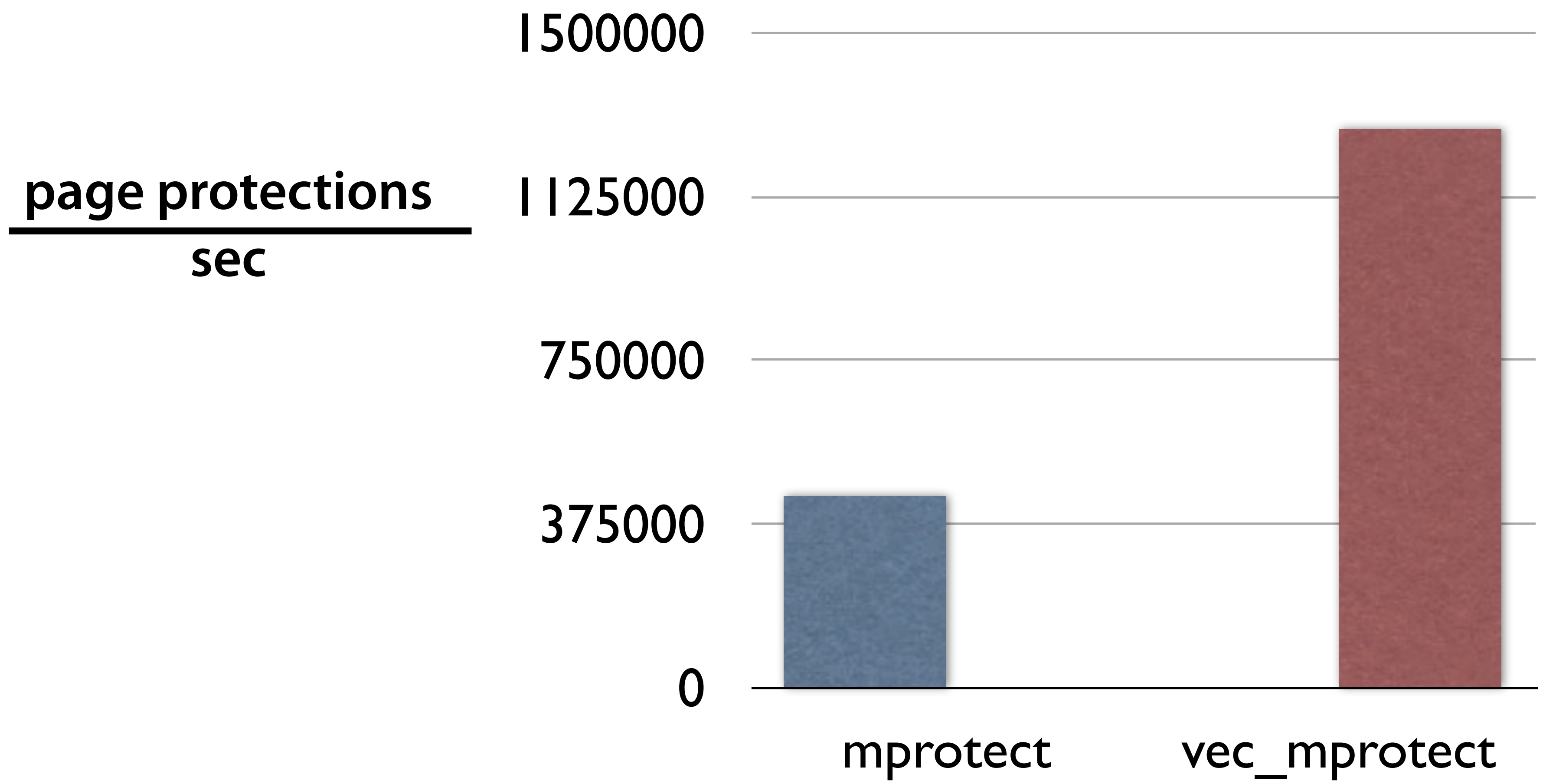


vec_mprotect techniques:

- Amortize context switches (async batching)

Data courtesy of Iulian Moraru

Speeding up memory protection

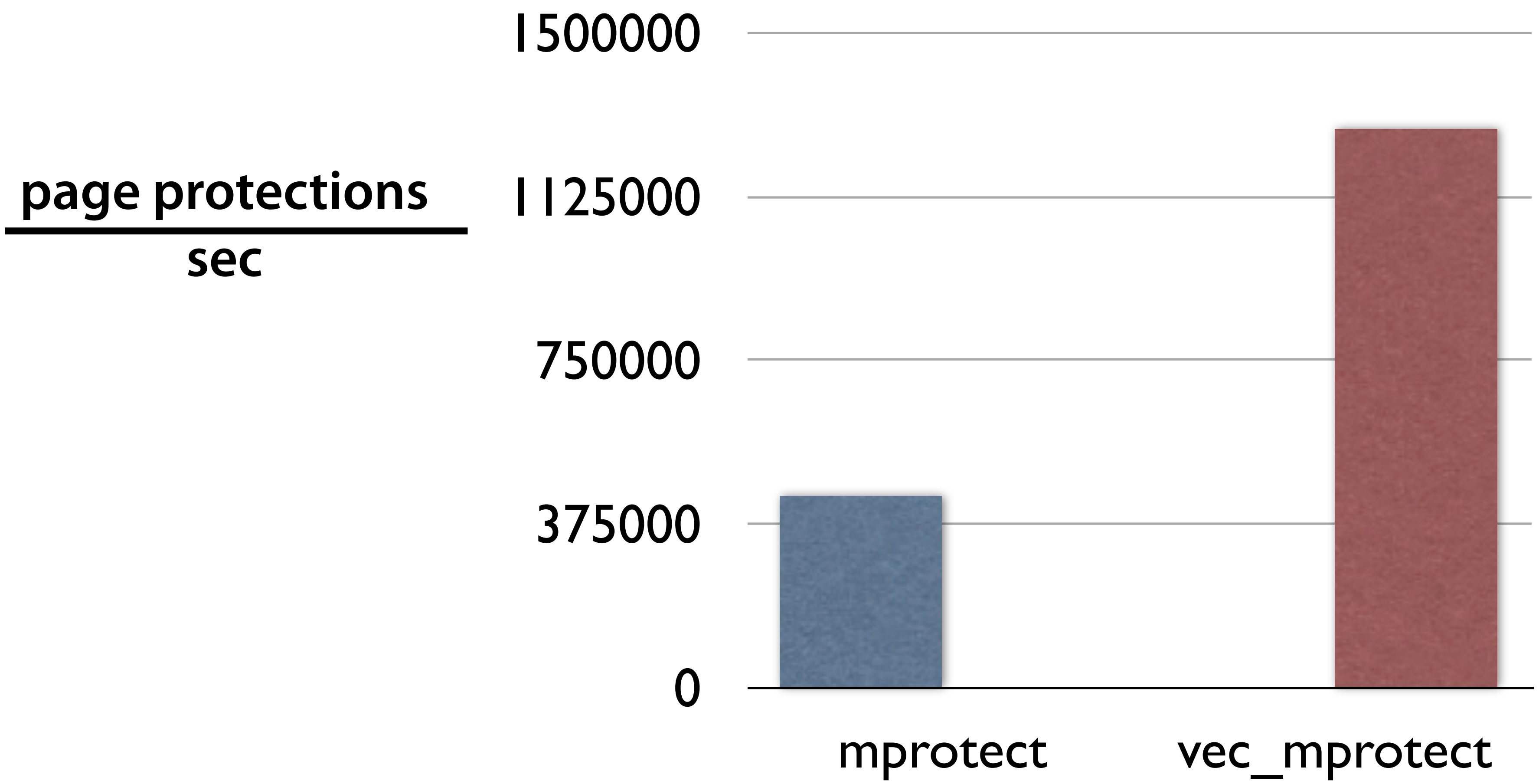


vec_mprotect techniques:

- Amortize context switches (async batching)
- Optimized data structure allocation (sorting)

Data courtesy of Iulian Moraru

Speeding up memory protection

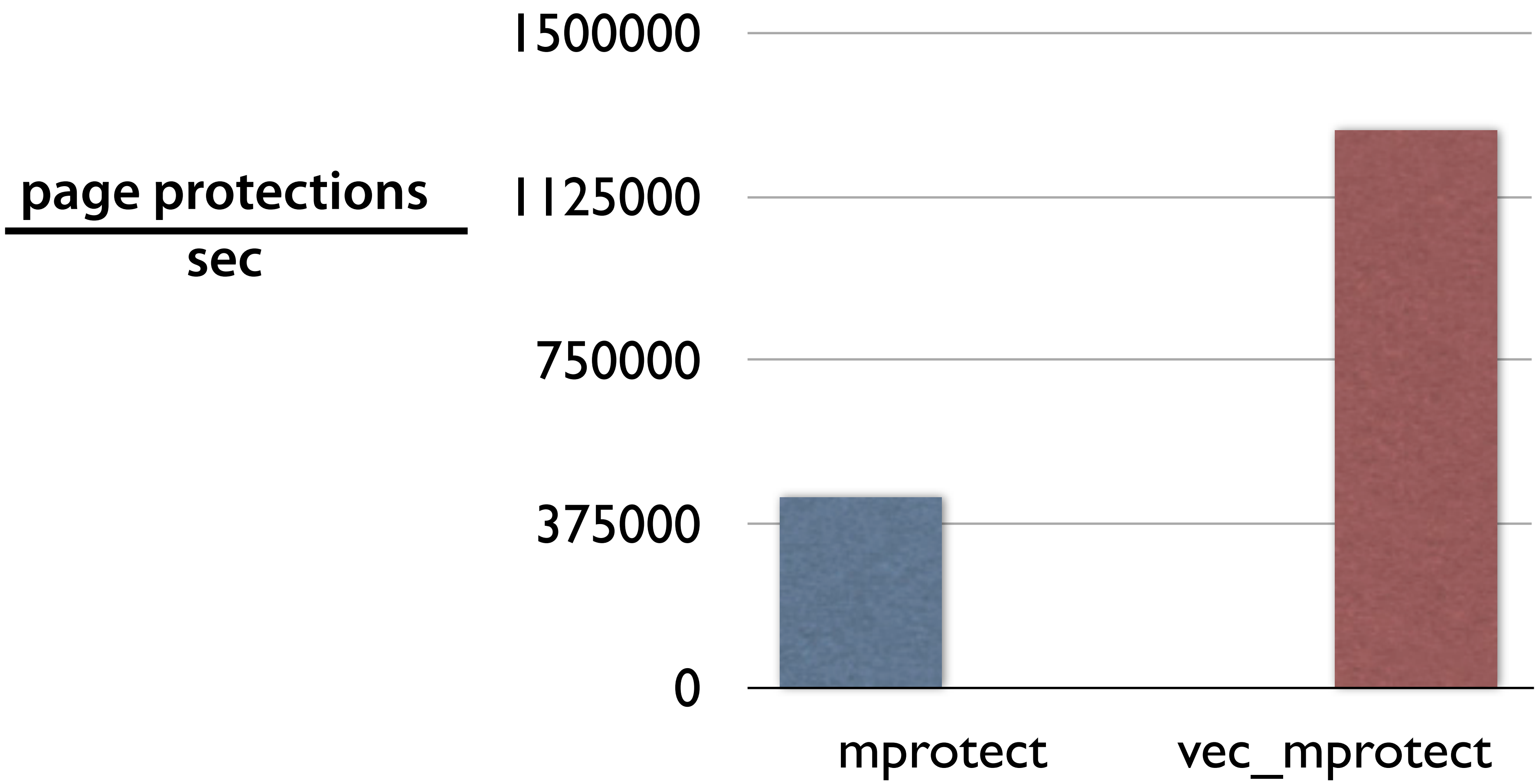


vec_mprotect techniques:

- Amortize context switches (async batching)
- Optimized data structure allocation (sorting)
- Eliminate TLB flush per individual call

Data courtesy of Iulian Moraru

Speeding up memory protection



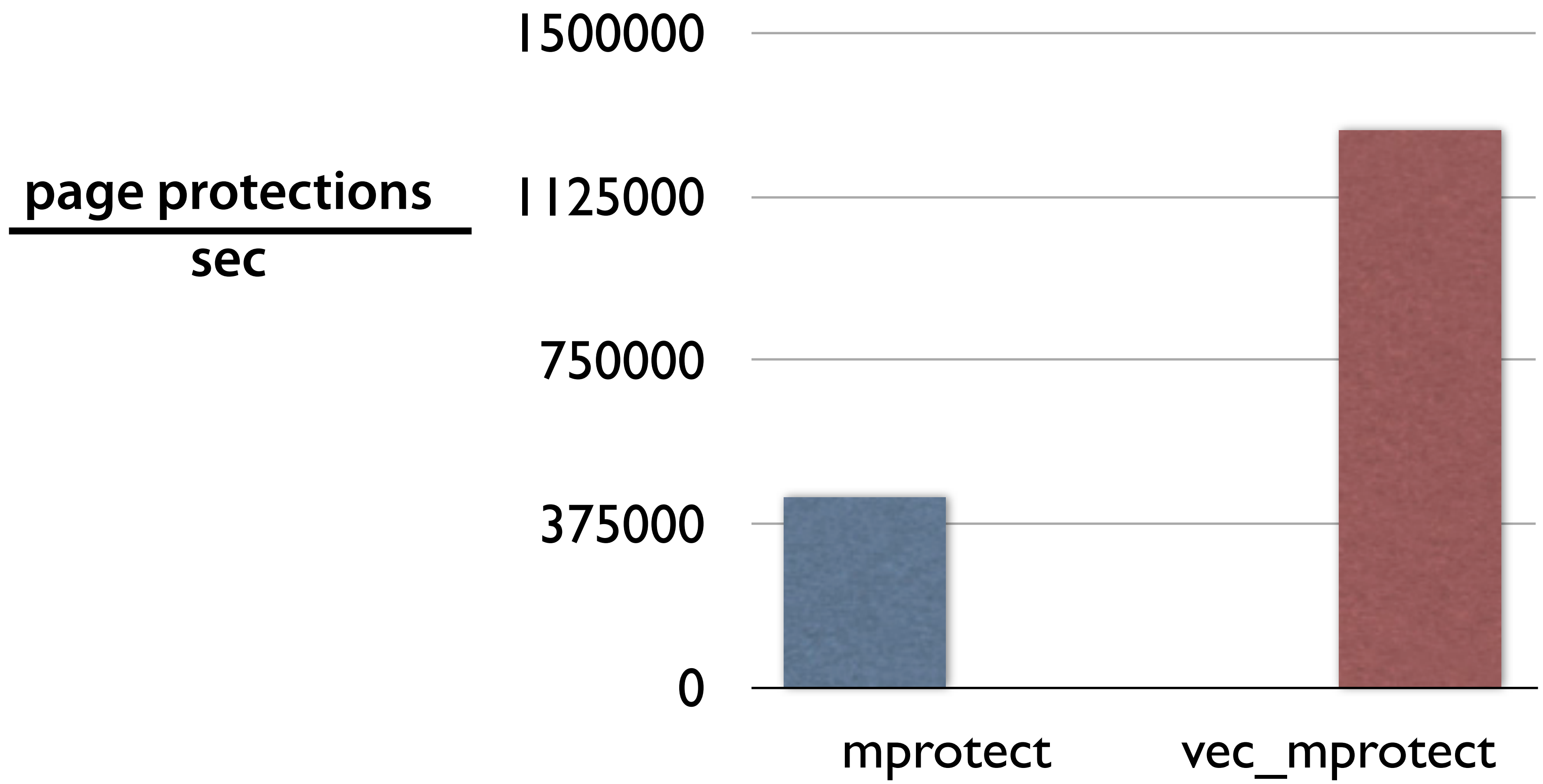
3x performance improvement

vec_mprotect techniques:

- Amortize context switches (async batching)
- Optimized data structure allocation (sorting)
- Eliminate TLB flush per individual call

Data courtesy of Iulian Moraru

Speeding up memory protection



3x performance improvement

vec_mprotect techniques:

- **30% {** Amortize context switches (async batching)
- **170% {** Optimized data structure allocation (sorting)
- Eliminate TLB flush per individual call

Data courtesy of Iulian Moraru

One difficult challenge

Handling convergence and divergence

```
vec_open([f1,f2,f3])
    context switch
    vec_alloc()
    vec_copy([f1,f2,f3])

    vec_path_resolve([f1])                      vec_path_resolve([f2,f3])
        acl_check([f1])                             acl_check([f2,f3])

                 h = hash([f1,f2,f3])

    lookup(h[0])                                lookup(h[1..2])
    read([f1])                                  read([f2,f3])

    dealloc()
    context switch
```

One difficult challenge

Handling convergence and divergence

```
vec_open([f1,f2,f3])
    context switch
    vec_alloc()
    vec_copy([f1,f2,f3])

    vec_path_resolve([f1])            vec_path_resolve([f2,f3])
        acl_check([f1])                   acl_check([f2,f3])

                 h = hash([f1,f2,f3])

    lookup(h[0])                      lookup(h[1..2])
    read([f1])                        read([f2,f3])

    dealloc()
    context switch
```

fork() ?

join() ?

One difficult challenge

Handling convergence and divergence

```
vec_open([f1,f2,f3])
    context switch
    vec_alloc()
    vec_copy([f1,f2,f3])

    vec_path_resolve([f1])                   vec_path_resolve([f2,f3])
        acl_check([f1])                          acl_check([f2,f3])

                 h = hash([f1,f2,f3])

    lookup(h[0])                             lookup(h[1..2])
    read([f1])                               read([f2,f3])

    dealloc()
    context switch
```

fork() ?

messages?

join() ?

One difficult challenge

Handling convergence and divergence

```
vec_open([f1,f2,f3])
    context switch
    vec_alloc()
    vec_copy([f1,f2,f3])
```

fork() ? **messages?**

```
    vec_path_resolve([f1])
        acl_check([f1])
```

```
vec_path_resolve([f2,f3])
    acl_check([f2,f3])
```

Is this worth joining for?

```
h = hash([f1,f2,f3])
```

```
    lookup(h[0])
    read([f1])
```

```
lookup(h[1..2])
read([f2,f3])
```

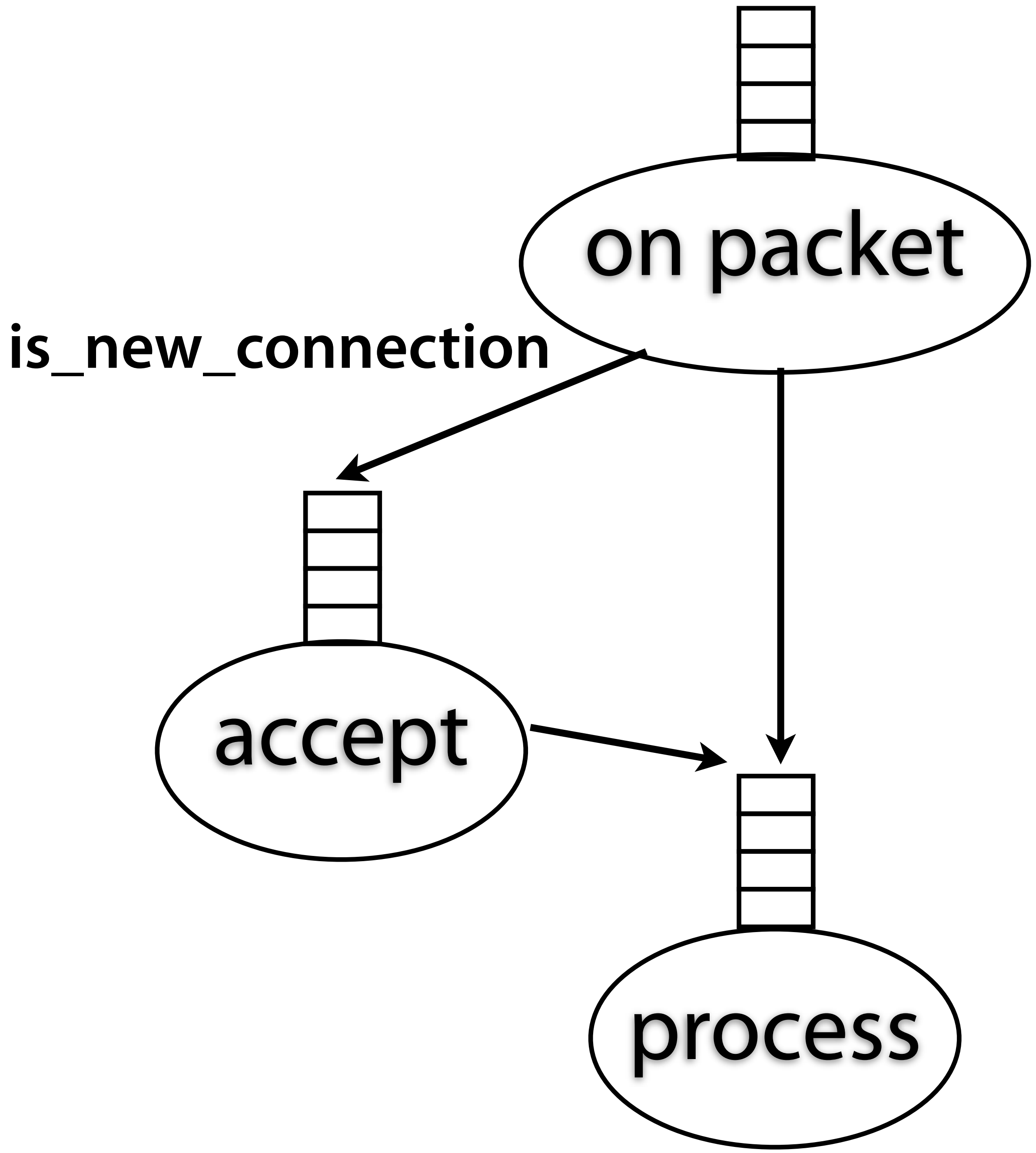
```
    dealloc()
    context switch
```

join() ?

OS as staged event system

Ideal interface for vectorization

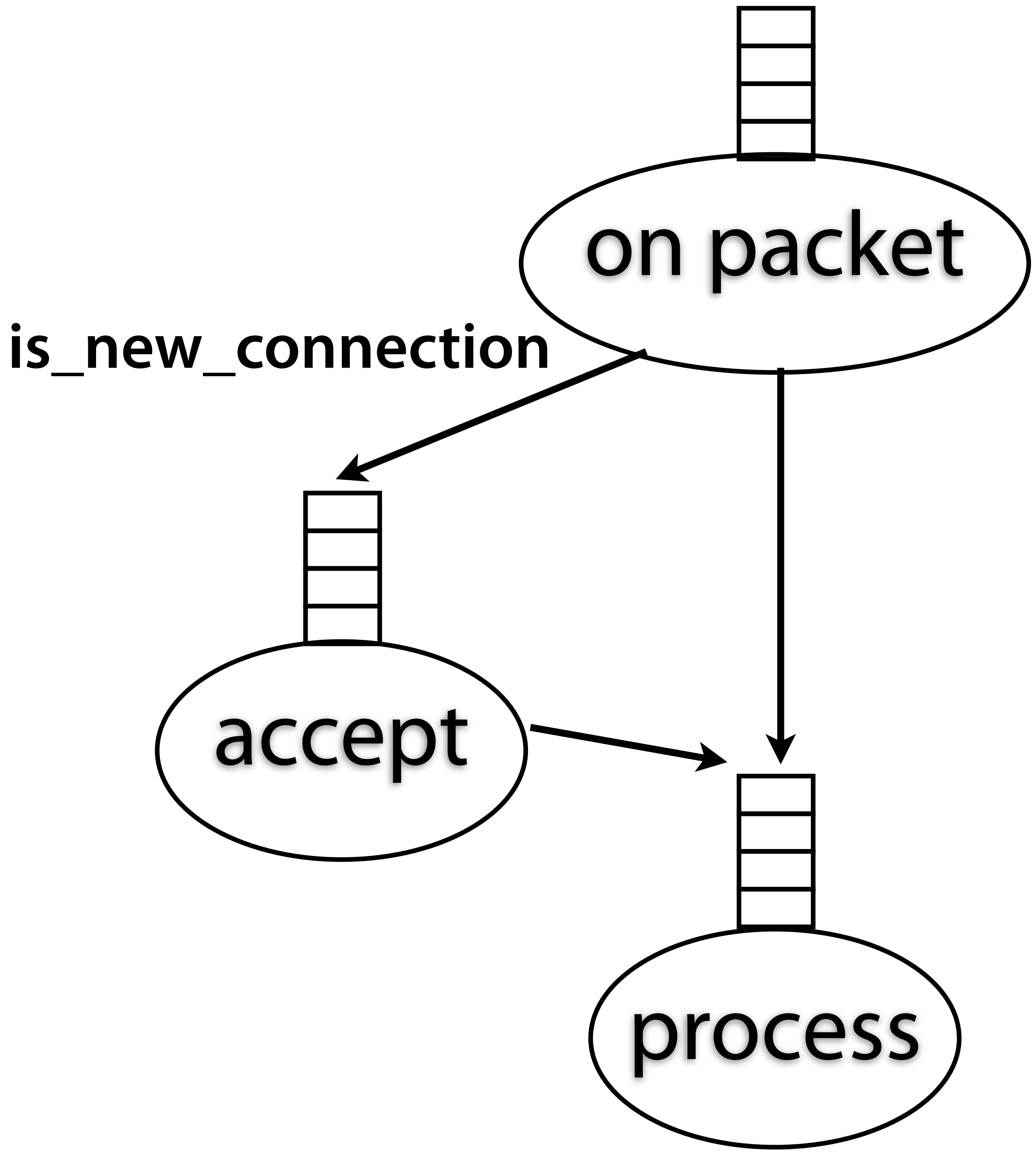
- Use message passing as underlying primitive



OS as staged event system

Ideal interface for vectorization

- Use message passing as underlying primitive



Programming interface?
Event abstraction is nice

Who vectorizes?
Static analysis, compiler
OS or App developer?

Efficiency vs. Latency

Summary of VOS

Vectors fundamentally improve efficiency by

- Collecting similar requests
- Eliminating redundant work
- Remaining parallel when code diverges

Challenges

- Programming vector abstractions
- Identifying what to coalesce and how to diverge

Summary of VOS

Don't let embarrassingly parallel
become embarrassingly wasteful

Vectors fundamentally improve efficiency by

- Collecting similar requests
- Eliminating redundant work
- Remaining parallel when code diverges

Challenges

- Programming vector abstractions
- Identifying what to coalesce and how to diverge

Related ideas

Community	Idea	Reason
HPC	Multicollective I/O readx/writex group open	I/O coalescing Reduced synch
Databases	Work Sharing Query Optimization	Reuse “results”, better I/O sched
OS	FlexSC Cassyopia, Cosy	Batching (all) system calls