RadixVM: Scalable address spaces for multithreaded applications

Austin T. Clements
M. Frans Kaashoek
Nickolai Zeldovich

MIT CSAIL
Parallel applications use VM intensively

RadixVM: Scalable address spaces for multithreaded applications
Parallel applications use VM intensively.

RadixVM: Scalable address spaces for multithreaded applications.

Every popular operating system serializes basic VM operations like mmap and munmap.
Parallel applications use VM intensively

Every popular operating system serializes basic VM operations like mmap and munmap.

RadixVM: Scalable address spaces for multithreaded applications
Application performance suffers

Multithreaded MapReduce [Mao '10]

Total throughput (jobs/hour)

# cores

RadixVM: Scalable address spaces for multithreaded applications
Inside parallel applications

Independent VM operations on non-overlapping regions.
Inside parallel applications

Independent VM operations on non-overlapping regions.

Common pattern for parallel applications.
Perfectly scalable mmap, munmap, and page fault operations on non-overlapping address space regions.
Structure of a VM system

Memory map

/sbin/ls

(s r w x)

(anonymous)

Hardware page table

Per-CPU TLB

<table>
<thead>
<tr>
<th>Virt</th>
<th>Phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>18bca</td>
<td>00230</td>
</tr>
<tr>
<td>87c38</td>
<td>0049c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virt</th>
<th>Phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>8a4bd</td>
<td>00382</td>
</tr>
<tr>
<td>87c38</td>
<td>0049c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virt</th>
<th>Phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>b987a</td>
<td>00520</td>
</tr>
<tr>
<td>8a4bd</td>
<td>00382</td>
</tr>
<tr>
<td>87c38</td>
<td>0049c</td>
</tr>
</tbody>
</table>
RadixVM: Scalable address spaces for multithreaded applications

- Structure of a VM system
- Memory map: Tracks mapped regions and region metadata
- Hardware page table
- Per-CPU TLB

Virt | Phys
---|---
18bca | 00230
87c38 | 0049c
8a4bd | 00382
87c38 | 0049c
b987a | 00520
8a4bd | 00382
87c38 | 0049c
Structure of a VM system

Memory map

Shared by OS and hardware. Maps virtual to physical.

Hardware page table

Per-CPU TLB

RadixVM: Scalable address spaces for multithreaded applications
Structure of a VM system

RadixVM: Scalable address spaces for multithreaded applications

Memory map

/bin/ls

(s r w x)

 anon

(s r w x)

Virt Phys
18bca 00230
87c38 0049c
8a4bd 00382
87c38 0049c
b987a 00520
8a4bd 00382

Caches page tables.
Internal to CPU.
Structure of a VM system

RadixVM: Scalable address spaces for multithreaded applications
Structure of a VM system

RadixVM: Scalable address spaces for multithreaded applications
Structure of a VM system

RadixVM: Scalable address spaces for multithreaded applications
Seems reasonable. Why doesn't it scale?
Structure of a VM system

Seems reasonable. Why doesn't it scale?

Locking
Structure of a VM system

Seems reasonable. Why doesn't it scale?

Shootdown broadcast  Locking

RadixVM: Scalable address spaces for multithreaded applications
Seems reasonable. Why doesn't it scale?
Shootdown broadcast  Locking  Cache contention
Structure of a VM system

Seems reasonable. Why doesn't it scale?

Shootdown broadcast Locking Cache contention

Cross-core communication
To achieve perfectly scalable non-overlapping operations, we eliminate communication between such operations.

Concurrent memory map representation

Method of targeting TLB shootdowns

Scalable, space-efficient reference counting
Metadata management

Need to store OS-level memory mapping metadata
Metadata management

Need to store OS-level memory mapping metadata

Popular operating systems use a balanced tree of region objects.
Need to store OS-level memory mapping metadata

Popular operating systems use a balanced tree of region objects. Memory-efficient
Metadata management

Need to store OS-level memory mapping metadata

Popular operating systems use a balanced tree of region objects.

- Unnecessary communication
- Memory-efficient communication
Need to store OS-level memory mapping metadata

Popular operating systems use a balanced tree of region objects.

Unnecessary communication

Memory-efficient communication
Need to store OS-level memory mapping metadata

Popular operating systems use a balanced tree of region objects.

Unnecessary communication

Memory-efficient communication

~1000 cycles
Need to store OS-level memory mapping metadata

Popular operating systems use a balanced tree of region objects.

Unnecessary communication

Memory-efficient

Most potential data structures (skip lists, B-trees, etc.) induce communication between disjoint operations.

RadixVM: Scalable address spaces for multithreaded applications
Array-based memory map
Array-based memory map
Array-based memory map

$0$ to $2^{35}$
Array-based memory map

s r w x file

```
0
```

```
(anon) (anon) (anon) (anon) (anon)
```

```
/lib/libc /lib/libc /lib/libc /lib/libc /lib/libc
```

$2^{35}$
### Array-based memory map

<table>
<thead>
<tr>
<th>s</th>
<th>r</th>
<th>w</th>
<th>x</th>
<th>file</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(anon)</td>
<td>(anon)</td>
<td>(anon)</td>
<td>(anon)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Good: Operations on non-overlapping regions are concurrent and induce no communication.
Array-based memory map

Good: Operations on non-overlapping regions are concurrent and induce no communication.

Bad: Space use is obscene, time is proportional to region size.
Array-based memory map

Good: Operations on non-overlapping regions are concurrent and induce no communication.

Bad: Space use is obscene, time is proportional to region size

How can we achieve good concurrency while keeping space and time under control?
Solution: Range-oriented radix tree
Radix tree

Solution: Range-oriented radix tree

RadixVM: Scalable address spaces for multithreaded applications
Radix tree

Solution: Range-oriented radix tree

RadixVM: Scalable address spaces for multithreaded applications
Radix tree

Solution: Range-oriented radix tree
Solution: Range-oriented radix tree

Fold constant-valued chunks into parent, recursively.
Radix tree

RadixVM: Scalable address spaces for multithreaded applications

Solution: Range-oriented radix tree

Fold constant-valued chunks into parent, recursively.
Solution: Range-oriented radix tree

Fold constant-valued chunks into parent, recursively.
Radix tree

Solution: Range-oriented radix tree

Fold constant-valued chunks into parent, recursively.

2-3x the size of the balanced region tree
Radix tree

Solution: Range-oriented radix tree

Fold constant-valued chunks into parent, recursively.

2-3x the size of the balanced region tree

We can achieve array-like concurrency with time and space similar to the balanced tree.
munmap must notify cores of changes to cached mappings
munmap must notify cores of changes to cached mappings

Which cores have a mapping cached? Who knows?!
munmap must notify cores of changes to cached mappings
Which cores have a mapping cached? Who knows?!
TLB shootdown

munmap must notify cores of changes to cached mappings

Which cores have a mapping cached? Who knows?!
munmap must notify cores of changes to cached mappings

Which cores have a mapping cached? Who knows?!
munmap must notify cores of changes to cached mappings

Which cores have a mapping cached? Who knows?!

In the common case, there is little or no sharing.
A software-managed TLB would make this easy.
A software-managed TLB would make this easy.

Trap and track
A software-managed TLB would make this easy.
Soft TLBs, the hard way

Solution: Per-core page tables for precise TLB tracking

Page faults

Page faults

TLB misses

TLB misses

TLB misses

Virt Phys
18bca 00230
87c38 0049c

Virt Phys
8a4bd 00382
87c38 0049c

Virt Phys
b987a 00520
8a4bd 00382
87c38 0049c

RadixVM: Scalable address spaces for multithreaded applications
Soft TLBs, the hard way

Solution: Per-core page tables for precise TLB tracking

RadixVM: Scalable address spaces for multithreaded applications
Soft TLBs, the hard way

Solution: Per-core page tables for precise TLB tracking

Trap and track
Soft TLBs, the hard way

Solution: Per-core page tables for precise TLB tracking

TLB tracking allows us to target TLB shootdowns, eliminating unnecessary shootdown communication.

Trap and track
Reference counting for physical pages and radix nodes
Reference counting for physical pages and radix nodes

Shared counters

Scalable inc/dec

N
Reference counting

Reference counting for physical pages and radix nodes

<table>
<thead>
<tr>
<th></th>
<th>Shared counters</th>
<th>Distributed counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalable inc/dec</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Zero-detection cost</td>
<td>O(1)</td>
<td>O(objs*cpus)</td>
</tr>
<tr>
<td>Space</td>
<td>O(1)</td>
<td>O(cpus)</td>
</tr>
</tbody>
</table>
Reference counting

Reference counting for physical pages and radix nodes

<table>
<thead>
<tr>
<th></th>
<th>Shared counters</th>
<th>Distributed counters</th>
<th>SNZIs [Ellen '07]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalable inc/dec</td>
<td>N</td>
<td>Y</td>
<td>Mostly</td>
</tr>
<tr>
<td>Zero-detection cost</td>
<td>O(1)</td>
<td>O(objs*cpus)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Space</td>
<td>O(1)</td>
<td>O(cpus)</td>
<td>O(cpus)</td>
</tr>
</tbody>
</table>
### Reference counting for physical pages and radix nodes

<table>
<thead>
<tr>
<th></th>
<th>Shared counters</th>
<th>Distributed counters</th>
<th>SNZIs [Ellen '07]</th>
<th>Refcache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalable inc/dec</td>
<td>N</td>
<td>Y</td>
<td>Mostly</td>
<td>Y</td>
</tr>
<tr>
<td>Zero-detection cost</td>
<td>O(1)</td>
<td>O(objs*cpus)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Space</td>
<td>O(1)</td>
<td>O(cpus)</td>
<td>O(cpus)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
## Reference counting

### Reference counting for physical pages and radix nodes

<table>
<thead>
<tr>
<th></th>
<th>Shared counters</th>
<th>Distributed counters</th>
<th>SNZIs [Ellen ’07]</th>
<th>Refcache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalable inc/dec</td>
<td>N</td>
<td>Y</td>
<td>Mostly</td>
<td>Y</td>
</tr>
<tr>
<td>Zero-detection cost</td>
<td>O(1)</td>
<td>O(objs*cpus)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Space</td>
<td>O(1)</td>
<td>O(cpus)</td>
<td>O(cpus)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Immediate zero detection</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

RadixVM: Scalable address spaces for multithreaded applications
Refcache

Approach: Shared counters with per-core delta caches
## Approach: Shared counters with per-core delta caches

<table>
<thead>
<tr>
<th>Single counter per object</th>
<th>Object A</th>
<th>Object B</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_count = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>global_count = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RadixVM: Scalable address spaces for multithreaded applications

Approach: Shared counters with per-core delta caches

Single counter per object

Object A
- global_count = 1
- ...

Object B
- global_count = 2
- ...

Caches changes, not values

<table>
<thead>
<tr>
<th>V</th>
<th>Object</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th>Object</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th>Object</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th>Object</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Refcache

Approach: Shared counters with per-core delta caches

Single counter per object

Object A

global_count = 1

...

Object B

global_count = 2

...

Caches changes, not values

V Object Delta

<table>
<thead>
<tr>
<th>0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU 0

V Object Delta

<table>
<thead>
<tr>
<th>0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>+1</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU 1 inc(A)

Operations are local

V Object Delta

<table>
<thead>
<tr>
<th>0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>+1</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU 2 inc(A)

V Object Delta

<table>
<thead>
<tr>
<th>0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU 3
Refcache

Approach: Shared counters with per-core delta caches

- Single counter per object
  - Object A: global_count = 1
  - Object B: global_count = 2

Caches changes, not values

- Operations are local
  - CPU 0: inc(A)
  - CPU 1: inc(A)
  - CPU 2: inc(A)
  - CPU 3: dec(B)

RadixVM: Scalable address spaces for multithreaded applications
Approach: Shared counters with per-core delta caches

Single counter per object

Object A

global_count = 1

... 

Object B

global_count = 2

... 

Caches changes, not values

True count = \( \sum \) 

Operations are local

CPU 0

V Object Delta

0 0 0 

CPU 1

V Object Delta

0 1 0 inc(A) +1

CPU 2

V Object Delta

0 1 0 inc(A) +1

CPU 3

V Object Delta

0 1 0 dec(B) -1

Generally unknown

RadixVM: Scalable address spaces for multithreaded applications
When is the true count zero?
When is the true count zero?

Assumption: When the true count is zero, it will stay zero.
When is the true count zero?

Assumption: When the true count is zero, it will stay zero.

Divide time into epochs. Each epoch, all CPUs flush their delta caches. If an object's global count stays zero for a whole epoch, then its true count is zero.
When is the true count zero?

Assumption: When the true count is zero, it will stay zero.

Divide time into epochs. Each epoch, all CPUs flush their delta caches. If an object's global count stays zero for a whole epoch, then its true count is zero.
When is the true count zero?

Assumption: When the true count is zero, it will stay zero.

Divide time in to epochs. Each epoch, all CPUs flush their delta caches. If an object's global count stays zero for a whole epoch, then its true count is zero.
Initially: Global count is 1, no cached deltas (so true count is 1)
Initially: Global count is 1, no cached deltas (so true count is 1)
Initially: Global count is 1, no cached deltas (so true count is 1)

CPU 0 decrements and flushes; global count is now 0. What about true count?
Initially: Global count is 1, no cached deltas (so true count is 1)

CPU 0 decrements and flushes; global count is now 0. What about true count?

CPU 1 increments after flush, before CPU 0's decrement
Initially: Global count is 1, no cached deltas (so true count is 1)

CPU 0 decrements and flushes; global count is now 0. What about true count?

CPU 1 increments after flush, before CPU 0's decrement

The true count is the sum of everything up to right now.
Initially: Global count is 1, no cached deltas (so true count is 1)

CPU 1 increments after flush, before CPU 0's decrement

CPU 0 decrements and flushes; global count is now 0. What about true count?

The true count is the sum of everything up to right now. But the global count only reflects the blue region. Operations in the orange region are still cached.
Initially: Global count is 1, no cached deltas (so true count is 1)

CPU 0 decrements and flushes; global count is now 0. What about true count?

CPU 1 increments after flush, before CPU 0's decrement

The true count is the sum of everything up to right now. But the global count only reflects the blue region. Operations in the orange region are still cached.
Initially: Global count is 1, no cached deltas (so true count is 1)

CPU 1 increments after flush, before CPU 0's decrement

CPU 0 decrements and flushes; global count is now 0. What about true count?

Global count now reflects cached ops

The true count is the sum of everything up to right now. But the global count only reflects the blue region. Operations in the orange region are still cached.
Initially: Global count is 1, no cached deltas (so true count is 1)

CPU 1 increments after flush, before CPU 0's decrement

CPU 0 decrements and flushes; global count is now 0. What about true count?

Global count now reflects cached ops

The true count is the sum of everything up to right now. But the global count only reflects the blue region. Operations in the orange region are still cached.

Abort delete
Refcache example

Initially: Global count is 1, no cached deltas (so true count is 1)

CPU 0 decrements and flushes; global count is now 0. What about true count?

CPU 1 increments after flush, before CPU 0's decrement

Global count now reflects cached ops

Refcache enables time- and space-efficient scalable reference counting with minimal latency.

Operations in the orange region are still cached. Abort delete
Bringing it all together

RadixVM: Scalable address spaces for multithreaded applications
Bringing it all together

RadixVM: Scalable address spaces for multithreaded applications

Per-core page tables

Radix tree memory map

mmap

Reference counted physical pages
Bringing it all together

RadixVM: Scalable address spaces for multithreaded applications

Radix tree memory map

Per-core page tables

Reference counted physical pages
Bringing it all together

Per-core page tables

Radix tree memory map

Reference counted physical pages

Page fault

RadixVM: Scalable address spaces for multithreaded applications
Bringing it all together

RadixVM: Scalable address spaces for multithreaded applications

Page fault

Record faulting CPU

Allocate backing page

Install in local page table

Per-core page tables

Reference counted physical pages

Radix tree memory map

s r w x file

cores

(anon)

(anon)

(anon)

(anon)
Bringing it all together

RadixVM: Scalable address spaces for multithreaded applications

Radix tree memory map

Per-core page tables

Reference counted physical pages
Bringing it all together

RadixVM: Scalable address spaces for multithreaded applications
Bringing it all together

RadixVM: Scalable address spaces for multithreaded applications

Per-core page tables

Radix tree memory map

File cores

Reference counted physical pages
We built RadixVM in a custom research kernel.

We believe RadixVM could be built in a mainstream kernel.

All benchmarks are source-compatible with Linux.
The other 99% is perspiration

Booting 80 cores (ACPI, x2APIC, IOMMU, oh my!)

NUMA-aware everything (memory allocation, per-CPU data, etc)

Performance analysis tools (NMI profiling, PEBS, load latency profiling, statistics counters)

Hardware curve balls (false sharing, bad prefetch behavior, etc)

All necessary for good results; all standard engineering.
Does parallel mmap/munmap matter to applications?

Are all of RadixVM's components necessary for scalability?
RadixVM improves application scalability

Metis multicore MapReduce [Mao '10], inverse indexing application

- **Total throughput (jobs/hour)**
- **# cores**

- **RadixVM**
- **Linux 3.5.7**
RadixVM improves application scalability

Metis multicore MapReduce [Mao '10], inverse indexing application

Total throughput (jobs/hour)

- RadixVM
- Linux 3.5.7

Page fault lock contention

RadixVM: Scalable address spaces for multithreaded applications
RadixVM improves application scalability

Metis multicore MapReduce [Mao '10], inverse indexing application

Total throughput (jobs/hour) vs. # cores

- RadixVM
- Linux 3.5.7

Pairwise sharing
Page fault lock contention
Radix trees avoid communication.

RadixVM: Scalable address spaces for multithreaded applications.

- Lookup existing keys
- Insert/delete random keys

~No communication
Linear scalability

Graph:
- Y-axis: Total throughput (lookups/sec)
- X-axis: # cores (n/2 readers, n/2 writers)
- Green line: Radix tree
- Blue line: Lock-free skiplist

- Total throughput increases linearly with the number of cores.
- The green line (Radix tree) shows a linear increase with no communication.

RadixVM: Scalable address spaces for multithreaded applications.
Refcache avoids cache line sharing

![Graph showing total map-unmap pairs per second vs. number of cores]

- **Y-axis**: Total map-unmap pairs/sec
- **X-axis**: Number of cores

- **Legend**:
  - Green line: Refcache
  - Blue line: Shared counter

- **Subtitle**: RadixVM: Scalable address spaces for multithreaded applications
Targeted TLB shootdown improves scalability

RadixVM: Scalable address spaces for multithreaded applications
Targeted TLB shootdown improves scalability

Core-local address space use

- Per-core
- Shared

No TLB shootdowns

RadixVM: Scalable address spaces for multithreaded applications
Targeted TLB shootdown improves scalability

Core-local address space use

- Per-core
- Shared

No TLB

Global address space use

- Per-core
- Shared
- Page table contention

Per-core overhead

RadixVM: Scalable address spaces for multithreaded applications
Related work

Scalable VM systems
• K42 [Krieger '06]
• Corey [Boyd-Wickizer '08]
• Bonsai [Clements '12]

Scalable reference counters
• Modula-2+ local refs [DeTreville '90]
• Distributed counters [Appavoo '07]
• Scalable non-zero indicators [Ellen '07]
• Sloppy counters [Boyd-Wickizer '10]
RadixVM: Scalable address spaces for multithreaded applications
Radix trees Per-core page tables Refcache

Perfect scalability for non-overlapping VM operations
Radix trees  

Perfect scalability for non-overlapping VM operations

Check it out: http://pdos.csail.mit.edu/multicore