RadixVM: Scalable address spaces for multithreaded applications

Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

ABSTRACT

RadixVM is a new virtual memory system design that enables fully concurrent operations on shared address spaces for multithreaded processes on cache-coherent multicore computers. Today, most operating systems serialize operations such as mmap and munmap, which forces application developers to split their multithreaded applications into multiprocess applications, hoard memory to avoid the overhead of returning it, and so on. RadixVM removes this burden from application developers by ensuring that address space operations on non-overlapping memory regions scale perfectly. It does so by combining three techniques: 1) it organizes metadata in a radix tree instead of a balanced tree to avoid unnecessary cache line movement; 2) it uses a novel memory-efficient distributed reference counting scheme; and 3) it uses a new scheme to target remote TLB shootdowns and to often avoid them altogether. Experiments on an 80 core machine show that RadixVM achieves perfect scalability for non-overlapping regions: if several threads mmap or munmap pages in parallel, they can run completely independently and induce no cache coherence traffic.

1 INTRODUCTION

Multithreaded applications on many-core processors can be bottlenecked by contended locks inside the operating system's virtual memory system. Because of complex invariants in virtual memory systems, widely used kernels, such as Linux and FreeBSD, have a single lock per shared address space. Recent research shows how to run a page fault handler in parallel with mmap and munmap calls in the same address space [7], but still serializes mmap and munmap, which applications use to allocate and return memory to the operating system. However, if the mmap and munmap involve non-overlapping memory regions in the shared address space, as is the case in memory allocation and freeing, then in principle these calls should be perfectly parallelizable because they operate on different parts of the address space. This paper contributes a new virtual memory design that achieves this goal.

A typical virtual memory system supports three key operations: mmap to add a region of memory to a process's address space, munmap to remove a region of memory from the address space and the corresponding pages from the hardware page tables, and pagefault, which inserts a page into the hardware page tables at the faulting virtual address, allocating a new physical page if necessary. This paper focuses on workloads in which multiple threads running in the same address space frequently and concurrently invoke these operations. Many multithreaded memory-intensive applications fit this mold: mmap, munmap, and related variants often lie at the core of high-performance memory allocators and garbage collectors. Applications that frequently map and unmap files also generate such workloads for the OS kernel.

Because operating systems serialize mmap and munmap calls, even for non-overlapping memory regions, these applications can easily be bottlenecked by contention in the OS kernel. As a result, application developers often work around the virtual memory system to minimize or circumvent this bottleneck. Multithreaded memory allocators provide a rich set of workaround examples: they allocate memory in large chunks from the operating system in each thread [21], batch many munmaps [13], don't return memory at all [15], or provide a loadable kernel module that implements custom VM operations for the allocator [26]. Some workarounds have deep structural implications, such as implementing an application as communicating single-threaded processes instead of as a single multi-threaded process [6].

Frequently these workarounds suffice to avoid the internal VM bottleneck, but often come with other downsides. For example, a Google engineer reported to us that Google's memory allocator is reluctant to return memory to the OS precisely because of scaling problems with munmap and as a result applications tie up gigabytes of memory until they exit. This delays the start of other applications and can cause servers to be used inefficiently. Engineers from other companies have reported similar problems to us.

With increasing core counts, we believe these workarounds will become increasingly complicated and their downsides more severe, so this paper attacks the root cause of VM scalability problems. This paper explores the design of a virtual memory system in which virtual memory operations contend only if they operate on overlapping memory regions. This ensures that if two threads operate on distinct ranges of vir-

Eurosys'13 April 15-17, 2013, Prague, Czech Republic

tual memory, the VM system does not get in the way, and no scalability workarounds are necessary. In this case, virtual memory operations should scale perfectly with the number of cores. If the application operates on the same shared region from multiple threads, sharing state between the two threads is inevitable, but the shared state should be minimal and constrained to the cores using the shared memory region.

There are several challenges in designing a virtual memory system in which the performance of parallel mmaps and munmaps scales with the number of cores performing the operations. First, there are complex invariants between different parts of the virtual memory system. For example, when a thread munmaps a region, the kernel must ensure that this region is removed from the hardware page tables before reusing the physical memory; otherwise, other threads in the same process can access memory that the kernel may have immediately repurposed for some other application. To provide this semantics the kernel must enforce ordering on operations, which can limit performance on larger numbers of cores. Second, even a *single* contended cache line can become a scaling bottleneck with many cores. One of our initial designs used a lock-free concurrent skip list, but interior nodes in the skip list became contended and limited scalability (see §5). Third, shooting down the translation lookaside buffer (TLB) of other cores—to ensure that no core caches an out-of-date mapping—can become a scaling bottleneck. Fourth, many designs that avoid the scaling problems in a straightforward manner have a large memory overhead that grows with the number of cores.

This paper addresses these challenges in a new design, which we call RadixVM. RadixVM uses three different ideas that complement each other to enable munmap, mmap, and page faults on non-overlapping memory regions to scale perfectly. First, it uses a carefully designed radix tree to record mapped memory regions. Second, it uses a novel scalable reference counting scheme for tracking when physical pages are free and radix tree nodes are no longer used. Third, when a page must be unmapped, it avoids shooting down hardware TLBs that don't have that page mapping cached. The combination of these three ideas allows RadixVM to implement a straightforward concurrency plan that ensures the correctness of VM operations while allowing VM operations on non-overlapping memory regions to scale.

We have implemented RadixVM in a new research kernel derived from xv6 [9] because changing the Linux virtual memory implementation is very labor intensive owing to its overall complexity and how interconnected it is with other kernel subsystems [7]. An experimental evaluation on an 80 core machine with microbenchmarks that represent common sharing patterns in multithreaded applications and a parallel MapReduce library from MOSBENCH [6] show that the RadixVM design scales for non-overlapping mmaps and munmaps. A detailed breakdown shows that all three components of RadixVM's design are necessary to achieve good scalability.

One downside of prototyping RadixVM on a research kernel instead of Linux is that it is difficult to run large, complex applications that traditionally place significant load on the virtual memory system, such as the garbage collectors in Oracle's Java VM. Furthermore, RadixVM faces a chicken-and-egg problem: many VM-intensive applications have already been forced to design around the limitations of traditional VM systems. As a result, our experimental evaluation focuses on measuring RadixVM's performance under microbenchmarks that capture sharing patterns we have observed in VM-heavy workloads on Linux.

The rest of the paper is organized as follows. §2 discusses the related work. §3 presents the design of RadixVM. §4 summarizes our implementation. §5 evaluates RadixVM's performance, and §6 concludes.

2 RELATED WORK

RadixVM's design builds on research in 5 different areas: SMP Unix kernels, kernels designed for scalability, VM data structures, TLB shootdown schemes, and scalable reference counters. In each of these areas RadixVM makes a contribution, as we discuss in turn.

Unix kernels. The VM designs for Unix kernels and Windows typically use a single read-write lock to protect a shared address space, perhaps augmented with finer-grained locks for operations on individual memory regions [1, 24, 27]. The single read-write lock protects the OS index data structures that map virtual addresses to metadata for mapped memory regions, as well as invariants between the OS index data structure and the hardware page tables and TLBs. In our earlier work on the Bonsai VM system, we described a lock-free design for page faults in Linux, but that design still required the use of Linux's address space lock to serialize mmap and munmap operations [7].

Scalable kernels. Quite a number of kernels have been designed with the primary goal of scaling operating system services to many cores, including K42 [20] and Tornado [14], as well as more recent systems such as Barrelfish [3], Corey [5], and fos [30]. K42 and Tornado use clustered objects to represent an address space, allowing a region list for an address space to be replicated per processor. This design allows for concurrent mmaps by threads in the same address space, at the expense of munmaps, which need to contact every processor to update its local region list [14]. The RadixVM design allows *both* mmaps and munmaps to run in parallel.

The Barrelfish and fos multikernel designs don't support shared hardware page tables and index structures between kernel instances, and thus avoid their associated scaling bottlenecks. Both kernels support multithreaded user-level application that share memory, but require an expensive two-phase distributed TLB shootdown protocol to unmap memory (see below).

Corey introduces the notion of an address range, which allows an application to selectively share parts of its address space, instead of being forced to make an all-or-nothing decision. RadixVM doesn't expose this control to applications but could benefit from allowing applications to control the tradeoffs between per-core page tables and shared page tables (see §5).

VM data structures. One reason that widely used operating systems use a lock on the address space is that they use complex index data structures to guarantee $O(\log n)$ lookup time when a process has many mapped memory regions. Linux uses a red-black tree for the regions [27], FreeBSD uses a splay tree [1], and Solaris and Windows use AVL trees [24, 29]. Because these data structures require rebalancing when a memory region is inserted, they protect the entire data structure with a single lock.

The Bonsai balanced tree allows for lock-free lookups but serializes inserts and deletes in the Bonsai VM system [7]. Similarly, Howard and Walpole [18] propose a relativistic red-black tree that supports lock-free lookups and which they have extended to support concurrent inserts and deletes using software transactional memory [17]. They don't describe, however, how to apply this design to a virtual memory system, which needs to maintain invariants beyond the structural correctness of the region tree (e.g., munmap must clear page tables and invalidate TLBs atomically with removing the region from the tree). RadixVM opts instead for a radix-tree-based data structure, which allows for concurrent non-overlapping lookups, inserts, and deletes without the need for software transactional memory. Furthermore, precise range locking makes it possible to maintain cross-structure invariants.

TLB shootdown. In a multithreaded application, when a thread removes a region from its address space, the operating system must send shootdown interrupts to other processors to ensure that those processors flush translations for that region from their TLBs. Since these interrupts are expensive, many operating systems implement a scheme that allows these interrupts to be sent in parallel and to be batched [4]. The Barrelfish operating system uses a distributed two-phase TLB shootdown scheme (similar to Uhlig's [28]), and exploits a software broadcast tree to deliver the interrupts efficiently [3].

The main contribution of RadixVM's scheme is to limit the number of cores that must be contacted to perform the shootdown. x86 processors do not inform the kernel which TLBs have which memory mappings cached, and therefore most kernels conservatively send interrupts to all cores running the application. RadixVM uses a scheme that precisely tracks which cores may have each page mapping cached in their TLBs, which allows RadixVM to send TLB shootdown interrupts to just those cores and to entirely eliminate shootdowns for mappings that are not shared between cores.

Scalable reference counters. RadixVM's scheme for reference counting, Refcache, inherits ideas from sloppy counters [6], Scalable NonZero Indicators (SNZI) [12], distributed counters [2], shared vs. local counts in Modula-2+ [11], and approximate counters [8]. All of these techniques speed up increment and decrement using per-core counters, and require significantly more work to find the true total value. Refcache is a scalable counter approach specialized for reference counts where the true value needs to be known only when it reaches zero. Unlike sloppy, SNZI, distributed, and approximate counters, it doesn't require space proportional to the number of cores per counter. Like sloppy counters, Refcache's design intentionally delays and batches zero detection to reduce cache contention.

3 DESIGN

Achieving scalability for VM operations in *different* processes is easy since these operations involve per-process data structures. RadixVM's design is novel because it allows threads of the *same* process to perform mmap, munmap, and pagefault operations for non-overlapping memory regions in parallel. That is, if n threads in the same process allocate more memory, then these mmaps scale perfectly (they take the same amount of time to execute regardless of n). Similarly, if a thread on one core allocates memory and another thread in the same process on another core returns a different memory region to the operating system, then these operations do not slow each other down or interfere with any other mmaps or munmaps for different memory regions. Finally, if a core runs pagefault for an address while other cores are performing operations on regions that do not include that page, then RadixVM scales perfectly. On the other hand, if an mmap and munmap involve overlapping memory regions, or a thread faults on a page that is being concurrently mmaped or munmaped, RadixVM serializes those operations. The scalability of this design is easy for application developers to understand and take advantage of: applications should simply avoid concurrent manipulation of overlapping memory regions when possible.

To achieve perfect scalability on modern multicore computers, RadixVM strives to ensure that cores don't contend for *any* cache lines when operating on non-overlapping regions. On modern cache-coherent hardware, any contended cache line can be a scalability risk because frequently written cache lines must be re-read by other cores, an operation that typically serializes at the cache line's home node.

This section describes how RadixVM achieves perfect scalability for operations on non-overlapping regions. We first present the three key data structures that form the core of RadixVM and then describe how RadixVM uses these to implement standard VM operations.

3.1 Reference counting with Refcache

Reference counting is critical to many OS functions, and RadixVM is no different. Since two virtual memory regions

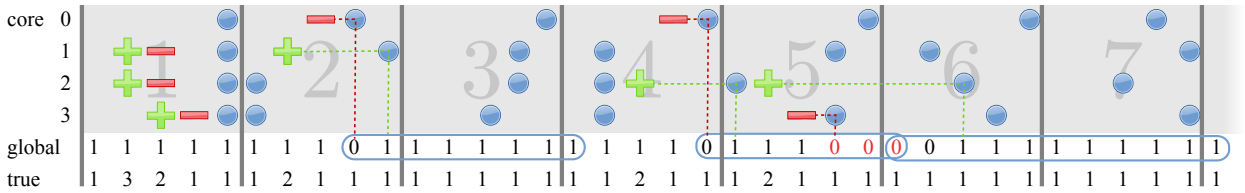

Figure 1: Refcache example showing a single object over eight epochs. Plus and minus symbols represent increment and decrement operations, dotted lines show when cores flush these to the object's global count, and blue circles show when each core flushes its local reference cache. The loops around the global count show when the object is in core 0's review queue and the red zeroes indicate dirty zeroes.

may share the same physical pages, such as when forking a process, RadixVM must have a way to decide when to free the underlying physical pages. To do so, RadixVM reference counts each physical page, but a simple scheme with a single counter can result in scalability problems because threads will contend for the counter. Likewise, RadixVM reference counts nodes of its radix tree to determine when they are empty; a single counter would cause operations on different parts of the node to contend.

This section introduces Refcache, a novel reference counting scheme that RadixVM uses to track and reclaim physical memory pages and radix tree nodes. Refcache implements space-efficient, lazy, scalable reference counting using per-core *reference delta caches*. Refcache targets workloads that can tolerate some latency in reclaiming resources and where increment and decrement operations often occur on the same core (e.g., the same thread that faulted pages into a mapped memory region also unmaps that region).

In contrast with most scalable reference counting mechanisms (see §2), Refcache requires space proportional to the sum of the number of reference counted objects and the number of cores, rather than the product, and the per-core overhead can be adjusted to trade off space and scalability by controlling the reference delta cache conflict rate. This is important when tracking every physical page in a large multicore system; at large core counts, typical scalable reference counters would require more than half of physical memory just to track the remaining physical memory.

Refcache batches increment and decrement operations, reducing cache line movement while offering an adjustable time bound on when an object will be garbage collected after its reference count drops to zero. Objects that are manipulated from only a single core do not require any per-object cache line movement and Refcache itself requires only a small constant rate of cache line movement for global maintenance.

Base Refcache. In Refcache, each reference counted object has a global reference count (much like a regular reference count) and each core also maintains a local, fixed-size cache of deltas to objects' reference counts. Incrementing or decrementing an object's reference count modifies only the local, cached delta and this delta is periodically flushed to the object's global reference count. The true reference count of an object is thus the sum of its global count and any local deltas for that object found in the per-core delta caches. The value of the true count is generally unknown, but we assume that once it drops to zero, it will remain zero (in the absence of weak references, which we discuss later). Refcache depends on this stability to detect a zero true count after some delay.

To detect a zero true reference count, Refcache divides time into periodic *epochs* during which each core flushes all of the reference count deltas in its cache, applying these updates to the global reference count of each object. The last core in an epoch to finish flushing its cache ends the epoch and all of the cores repeat this process after some delay (our implementation uses 10ms). Since these flushes occur in no particular order and the caches batch reference count changes, updates to the reference count can be reordered. As a result, a zero global reference count does not imply a zero true reference count. However, once the true count *is* zero, there will be no more updates, so if the global reference count of an object drops to zero and *remains* zero for an entire epoch, then Refcache can guarantee that the true count is zero and free the object. To detect this, the first core that sets an object's global reference count to zero adds the object to a per-core *review queue* and reexamines it two epochs later (which guarantees one complete epoch has elapsed) to decide whether its true reference count is zero.

Figure 1 gives an example of a single object over the course of eight epochs. Epoch 1 demonstrates the power of batching: despite six reference count manipulations spread over three cores, the object's global reference count is never written to. The remaining epochs demonstrate the complications that arise from batching and the resulting lag between the true reference count and the global reference count of an object.

Because of the flush order, the two updates in epoch 2 are applied to the global reference count in the opposite order of how they actually occurred. As a result, core 0 observes the global count temporarily drop to zero when it flushes in epoch 2, even though the true count is non-zero. This is remedied as soon as core 1 flushes its increment, and when core 0 reexamines the object at the beginning of epoch 4, after all cores have again flushed their delta caches, it can see that the global count is non-zero; hence, the zero count it observed was not a true zero and the object should not be freed.

It is not enough for the global reference count to be zero when an object is reexamined; rather, it must have been zero for the entire epoch. For example, core 0 will observe a zero global reference count at the end of epoch 4, and again when it reexamines the object in epoch 6. However, the true count is not zero, and the global reference count was temporarily non-zero during the epoch. We call this a *dirty* zero and in this situation Refcache will queue the object to be examined again two epochs later, in epoch 8.

Weak references. As described, Refcache is well suited to reference counts that track the true number of references to an object, since there is no danger of the count going back up once the object becomes unreachable. However, operating systems often need untracked references to objects; for example, OS caches track objects that may be deleted at any time, and may even need to bring an object's reference count back up from zero. RadixVM's radix tree has similar requirements. To support such uses, we extend Refcache with *weak references*, which provide a tryget operation that will either increment the object's reference count (even if it has reached zero) and return the object, or will indicate that the object has already been deleted.

A weak reference is simply a pointer marked with a "dying" bit, along with a back-reference from the referenced object. When an object's global reference count initially reaches zero, Refcache sets the weak reference's dying bit. After this, tryget can "revive" the object by atomically clearing the dying bit and fetching the pointer value, and then incrementing the object's reference count as usual. When Refcache decides to free an object, it first atomically clears both the dying bit and the pointer in the weak reference. If this succeeds, it can safely delete the object. If this fails, it reexamines the object again two epochs later. In a race between tryget and deletion, which operation succeeds is determined by which clears the dying bit first.

Algorithm. The pseudocode for Refcache is given in Figure 2. Each core maintains a hash table storing its reference delta cache and the review queue that tracks objects whose global reference counts reached zero. A core reviews an object after two epoch boundaries have passed so it can guarantee that all cores have flushed their reference caches at least once.

All of the functions in Figure 2 execute with preemption disabled, meaning they are atomic with respect to each other on a given core, which protects the consistency of per-core data structures. Individual objects are protected by fine-grained locks that protect the consistency of the object's fields.

For epoch management, our current implementation uses a barrier scheme that tracks a global epoch counter, per-core epochs, and a count of how many per-core epochs have reached the current global epoch. This scheme suffices for our benchmarks, but more scalable schemes are possible, such

```
inc(obj):
  if local cache[hash(obj)].obj ≠ obj:
    evict(local cache[hash(obj)])
    local cache[hash(obj)] ← ⟨obj, 0⟩
  local cache[hash(obj)].delta += 1

tryget(weakref):
  do:
    ⟨obj, dying⟩ ← weakref
  while weakref.cmpxchng(⟨obj, dying⟩, ⟨obj, false⟩) fails
  if obj is not null:
    inc(obj)
  return obj

flush():
  evict all non-zero local cache entries and clear cache
  update the current epoch

evict(obj, delta):
  with obj locked:
    obj.refcnt ← obj.refcnt + delta
    if obj.refcnt = 0:
      if obj is not on any review queue:
        obj.dirty ← false
        obj.weakref.dying ← true
        add ⟨obj, epoch⟩ to the local review queue
      else:
        obj.dirty ← true

review():
  for each ⟨obj, objepoch⟩ in local review queue:
    if epoch < objepoch + 2: continue
    with obj locked:
      remove obj from the review queue
      if obj.refcnt ≠ 0:
        obj.weakref.dying ← false
      else if obj.dirty or obj.weakref.cmpxchng(⟨obj, true⟩,
          ⟨null, false⟩) fails:
        obj.dirty ← false
        obj.weakref.dying ← true
        add ⟨obj, epoch⟩ to the local review queue
      else:
        free obj
```

Figure 2: Refcache algorithm. Each core calls flush and review periodically. evict may be called by flush or because of a collision in the reference cache. dec is identical to inc except that it decrements the locally cached delta.

as the tree-based quiescent state detection used by Linux's hierarchical RCU implementation [23].

Discussion. Refcache trades latency for scalability by batching increment and decrement operations in per-core caches. As a result, except when there are conflicts in the reference delta cache, increment and decrement operations do not share cache lines with other cores and communication is necessary only when these caches are periodically reconciled. Furthermore, because Refcache uses per-core caches rather than per-core counts, it is more space-efficient than other scalable reference counting techniques. While not all uses of reference counting can tolerate Refcache's latency, its scalability and space-efficiency are well suited to the requirements of RadixVM.

3.2 Radix tree

At its core, an address space is a mapping from virtual addresses to metadata and physical memory. To achieve perfectly scalable non-overlapping address space operations, RadixVM needs a data structure that can track this mapping and support mmaping and munmaping ranges of virtual address space while avoiding contention between operations on disjoint regions.

One option to avoid contention is to avoid sharing altogether. For example, one could attempt to partition the address space of a process statically among the cores, and have each core manage its part of the address space. This option ensures that operations on non-overlapping memory regions that are in different partitions of the address space don't contend for cache lines because each region is in a different partition. The downside of this option is that static partitioning complicates sharing and requires the application developer to be aware of the partitioning, so that each thread manipulates regions that are in its partition. A more desirable solution is to use some shared data structure that allows symmetric operations from all threads to manage the shared address space, as all current VM systems do.

In particular, a data structure that supports lock-free operations—such as the Bonsai tree does for read operations [7]—seems promising, since it avoids cache line contention due to locks. Lock-free operation, however, doesn't imply *no* cache line contention. For example, insert and lookup operations for a lock-free concurrent skip list [16] can result in contention for cache lines storing interior nodes in the skip list—even when the lookup and insert involve different keys—because insert must modify interior nodes to maintain $O(\log n)$ lookup time. As we will show in §5, this read-write sharing scales badly with more cores, as more cores need to reread cache lines modified by unrelated operations on other cores.

Any balanced tree or similar data structure suffers from this unintended cache line contention. A (completely impractical) strawman solution is to represent a process's virtual memory by storing the metadata for each virtual page individually in a large linear array indexed by virtual page number. In this linear representation, mmap, munmap, and pagefault can lock and manipulate precisely the pages being mapped, unmapped, or faulted. VM operations on non-overlapping memory regions will access disjoint parts of the linear array and thus scale perfectly. The design presented in this section follows the same general scheme as this strawman design, but makes its memory consumption practical using a multilevel, compressed radix tree.

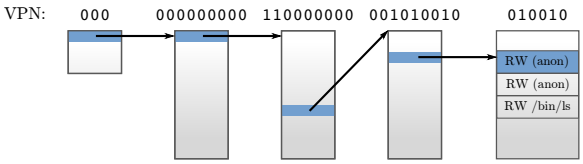

Figure 3: A radix tree containing both an anonymous mapping and a file mapping. Blue indicates the path for looking up the 36-bit virtual page number shown in bits at the top of the figure. The last level of the tree contains separate mapping metadata for each page.

The index data structure of RadixVM resembles a hardware page table structurally, storing mapping metadata in a fixed-depth radix tree, where each level of the tree is indexed by nine (or fewer) bits of the virtual page number (Figure 3). Like the linear array, the radix tree supports only point queries (not range queries) and iteration, but unlike the linear array, RadixVM can compress repeated entries and lazily allocate the nodes of the radix tree. Logically, any node that would consist entirely of identical values is folded into a single value stored in the parent node. This continues up to the root node of the tree, allowing the radix tree to represent vast swaths of unused virtual address space with a handful of NULL values and to set large ranges to identical values very quickly. This folding comes at a small cost: mmap operations that force expansion of the radix tree may conflict with each other, even if their regions do not ultimately overlap. However, such conflicts are rare.

To record each mapping, RadixVM stores a separate copy of the mapping metadata in the radix tree for each page in the mapped range. This differs from a typical design that allocates a single metadata object to represent the entire range of a mapping (e.g., virtual memory areas in Linux). Storing a separate copy of the metadata for each page makes sense in RadixVM because the metadata is relatively small, and eliminating shared objects avoids contention when a single mapping needs to be split or merged by a subsequent mmap or munmap call. Furthermore, the mapping metadata object is designed so that it will initially be identical for every page of a mapping, meaning that large mappings can created efficiently and folded into just a few slots in the radix tree's nodes.

Also unlike typical virtual memory system designs, RadixVM stores pointers to physical memory pages in the mapping metadata for pages that have been allocated. This

is easy to do in RadixVM because, modulo folding, there is a single mapping metadata object for each page. It's also important to have this canonical representation of the physical memory backing a virtual address space because of the way RadixVM handles TLB shootdown (see §3.3). This does increase the space required by the radix tree, but, asymptotically, it's no worse than the hardware page tables, and it means that the hardware page tables themselves are cacheable memory that can be discarded by the OS to free memory.

To keep the memory footprint of the radix tree in check, the OS must be able to free nodes that no longer contain any valid mapping metadata. To accomplish this without introducing contention, we leverage Refcache to scalably track the number of used slots in each node. When this count drops to zero, the radix tree can remove the node from the tree and delete it. Since RadixVM may begin using a node again before Refcache reconciles the used slot count, nodes link to their children using weak references, which allows the radix tree to revive nodes that go from empty to used before Refcache deletes them, and to safely detect when an empty child node has been deleted.

Collapsing the radix tree does introduce additional contention; however, in contrast with more eager garbage collection schemes, rapidly changing mappings cannot cause the radix tree to rapidly delete and recreate nodes. Since a node must go unused for at least two Refcache epochs before it is deleted, any cost of deleting or recreating it (and any additional contention that results) is amortized.

In contrast with more traditional balanced trees, using a radix tree to manage address space metadata allows RadixVM to achieve perfect scalability for operations on non-overlapping ranges of an address space. This comes at the cost of a potentially larger memory overhead; however, address space layouts tend to exhibit good locality and folding efficiently compresses large ranges, making radix trees a good fit for a VM system.

3.3 TLB shootdown

One complication with scaling mmap or munmap operations is the per-core hardware page translation cache, which requires explicit notifications ("TLB shootdowns") when a page mapping changes. Because TLB shootdowns must be delivered to every CPU that may have cached a page mapping that's being modified, and because hardware does not provide information about which CPUs may have cached a particular mapping, a conservative design must send TLB shootdown interrupts to all CPUs using the same address space, which limits scalability.

RadixVM achieves better scalability for mmap and munmap by keeping track of more precise information about the set of CPUs that may have accessed a given mapping, as part of the mapping metadata. With a software-filled TLB, the kernel can use TLB miss faults to track exactly which CPUs have a given mapping cached. When a later mmap or munmap changes this mapping, it can deliver shootdowns only to cores that have accessed this mapping. On architectures with hardware-filled TLBs such as the x86, our design achieves the same effect using per-core page tables. If a thread in an application allocates, accesses, and frees memory on one core, with no other threads accessing the same memory region, then RadixVM will perform no TLB shootdowns.

The obvious downside to this approach is the extra memory required for per-core page tables. We show in §5.4 that this overhead is small in practice compared to the total memory footprint of an application, but for applications with poor partitioning, it may be necessary for the application to provide hints about widely shared regions so the kernel can share page tables (similar to Corey address ranges [5]) or the kernel could detect such regions automatically. The kernel could also reduce overhead by sharing page tables between small groups of cores or by simply discarding page table pages when memory is low.

3.4 VM operations

The POSIX semantics of VM operations can make a scalable implementation of the operations challenging [7]. But, with the components described above, the RadixVM implementation of the VM operations is surprisingly straightforward. The POSIX semantics that are challenging are the ones that relate to ordering. For example, after the return of an munmap, no thread should be able to access the unmapped pages. Similarly, after mmap, every thread that experiences a pagefault should be able to access the mapped pages. There is also some complexity in pagefault because it is not just a read operation: it may have to allocate physical pages and modify the address space.

RadixVM primarily enforces correct ordering semantics by always locking, from left to right, the radix tree entries for the region affected by an operation. Each slot in the radix tree (in both interior and leaf nodes) reserves one bit for this purpose. As a result, two concurrent VM operations on overlapping ranges will serialize when locking the leftmost overlapping page.

When locking a region that has not been expanded out to leaf nodes yet, RadixVM acquires locks on the corresponding internal node slots instead. When RadixVM expands the tree by allocating new nodes, it propagates the lock bit to every entry in the newly allocated node, and unlocks the parent interior node slot. Releasing the lock clears the lock bits in the newly allocated child node. Tree traversal does not require locks because it increments each node's reference count through a Refcache weak reference.

An mmap invocation first locks the range being mapped. As above, if the leaf nodes for the range have already been allocated, mmap locks the mapping metadata in the leaf nodes, and if not, it locks the corresponding interior nodes. If there are existing mappings within the range, mmap unmaps them, as described later for munmap. mmap then fills in mapping

metadata for the new mapping (protection level and flags arguments to mmap, as well as what backs this virtual memory range, such as a file or anonymous memory). If the mapping covers an entire radix tree node, and the child nodes have not been allocated yet, the radix tree collapses the mapping metadata into a single slot in the interior of the tree. Otherwise, RadixVM copies the mapping metadata into each leaf node entry in the range. Finally, RadixVM unlocks the range. Like in other VM systems, mmap doesn't allocate any physical pages, but leaves that to pagefault, so that pages are allocated only when they are used.

A pagefault invocation traverses the radix tree to find the mapping metadata for the faulting address, and acquires a lock on it. It then allocates a physical page, if one has not been allocated yet, and stores it in the mapping metadata. Finally, pagefault fills in the page table entry in the local core's page table, and adds the local core number to the TLB shootdown list in the mapping metadata for that address. pagefault then releases the lock and returns.

To implement munmap, RadixVM must clear mapping metadata from the radix tree, clear page tables, invalidate TLBs, and free physical pages. munmap begins by locking the range being unmapped, after which it can scan the region's metadata to gather references to the physical pages backing the region, collect the set of cores that have faulted pages in the region into their per-core page tables, and clear each page's metadata. It can then send inter-processor interrupts to the set of cores it collected in the first step. These interrupts cause the remote cores (and the core running munmap) to clear the appropriate range in their per-core page table and invalidate the corresponding local TLB entries. Once all cores have completed this shootdown process, munmap can safely release its lock on the range and decrement the reference counts on the physical pages that were unmapped.

Note that an invocation of pagefault on one core may concurrently access a page that another core is in the process of unmapping, but the mapping metadata lock makes this work out correctly: either pagefault acquires the lock first or munmap does. In the first case, the page fault succeeds, which is okay since the pages must be inaccessible only after munmap returns. In the second case, the munmap runs first and removes the mapping for the unmapped range before releasing the locks. Then, pagefault will see that there is no mapping for the faulting address and will halt the faulting thread.

3.5 Discussion

By combining Refcache for scalable reference counting, radix trees for maintaining address space metadata, and per-core page tables for precise TLB tracking and shootdown, RadixVM is able to execute concurrent mmap, munmap, and pagefault operations on the same address space in a way that shares cache lines only when two operations manipulate overlapping regions and must be serialized. With the right data structures in place, RadixVM can achieve this with a straightforward concurrency plan based on precise range locking. We confirm that RadixVM's design translates into scalable performance for non-overlapping VM operations in §5.

Component	Line count
Radix tree	1,376
Refcache	932
MMU abstraction	889
Syscall interface	632

Table 1: Major RadixVM components.

4 IMPLEMENTATION

A challenge for our research is what system to use for evaluating RadixVM. On one hand it is natural to use the Linux kernel, since it is widely used, exhibits scaling problems because it serializes mmap and munmap operations (see §2), and would allow a completely fair comparison with unmodified Linux as well as with the Bonsai VM system, which was integrated with Linux [7]. On the other hand, Linux comprises millions of lines of code and modifications can require extraordinary amounts of work. We learned from our experience with the Bonsai VM system that modifying the Linux VM system for parallelizing read-only operations is difficult, and that modifying it for parallelizing write operations such as mmap and munmap is infeasible for a small group of researchers. Many different parts of the Linux kernel have their hands in the internals of the VM system.

For this paper, therefore, we use a small, POSIX-like monolithic kernel and operating system derived from xv6 [9]. This OS provides a POSIX-like API, hardware support for large multicore Intel and AMD 64-bit x86 processors, and a complete enough implementation of the standard C library that the benchmarks used in §5 can be compiled unmodified on both Linux and our OS. Our OS is written mostly in C++ so that we can create generic, reusable concurrent data structures (such as RadixVM's radix tree) and algorithms (such as Refcache's algorithm), as well as take advantage of C++11's features for multithreaded programming such as standard atomic types [19, §29]. Though our kernel lacks important features (e.g., swapping), we do not believe they will impact scalability of the basic POSIX VM operations.

Table 1 shows the lines of code for the major components of RadixVM. The implementation of RadixVM is 3,829 lines of code, and supports the full design described in §3, except for radix tree collapsing. The MMU code provides an abstract interface that is implemented both for per-core page tables, which provide targeted TLB shootdowns, and for traditional shared page tables. The RadixVM code relies heavily on generic abstractions provided by our kernel, such as basic data structures and synchronization primitives, which significantly reduces the size of its implementation.

5 EVALUATION

This section answers the following questions experimentally:

- Does RadixVM's design matter for applications?
- Why does RadixVM's design scale well?
- Is the memory overhead of RadixVM acceptable?
- Are all of RadixVM's design ideas necessary to achieve good scalability?

5.1 Experimental setup

To evaluate the impact of RadixVM on application performance, we use Metis, a high-performance single-server multithreaded MapReduce library, to compute a word position index from a 4 GB in-memory text file [10, 22]. Metis is a representative choice because it exhibits several virtual memory sharing patterns that are common in multithreaded applications: it uses core-local memory, it uses a globally shared B+-tree to store key-value pairs, and it also has pairwise sharing of intermediate results between Map tasks and Reduce tasks. Metis also allows a direct comparison with the Bonsai virtual memory system [7], which used Metis as its main benchmark.

By default Metis uses the Streamflow memory allocator [25], which is written to minimize pressure on the VM system, but nonetheless suffers from contention in the VM system when running on Linux [7]. Previous systems that used this library avoided contention for in-kernel locks by using super pages and improving the granularity of the super page allocation lock in Linux [6], or by having the memory allocator pre-allocate all memory upfront [22]. While these workarounds do allow Metis to scale on Linux, we wanted to focus on the root scalability problem in the VM system rather than the efficacy of workarounds and to eliminate compounding factors from differing library implementations, so we use a custom allocator on both Linux and RadixVM designed specifically for Metis. In contrast with modern memory allocators, this allocator is simple and designed to have no internal contention: memory is mapped in fixed-sized blocks, free lists are exclusively per-core, and the allocator never returns memory to the OS.

With this allocator, Metis stresses concurrent mmaps and pagefaults, but not concurrent munmaps. We explore other workloads using three microbenchmarks:

Local. Each thread mmaps a private region in the shared address space, writes to all of the pages in the region, and then munmaps its region. Many concurrent memory allocators use per-thread memory pools that specifically optimize for thread-local allocations and exhibit exactly this pattern of address space manipulation [13, 15]. However, such memory allocators typically map memory in large batches and conservatively return memory to the operating system to avoid putting pressure on the virtual memory system. Our microbenchmark does the opposite: it uses 4 KB regions to maximally stress the VM system.

Pipeline. Each thread mmaps a region of memory, writes to all of the pages in the region, and passes the region to the next thread in sequence, which also writes to all of the pages in the region, and then munmaps it. This captures the pattern of a streaming or pipelined computation, such as a Map task communicating with a Reduce task in Metis.

Global. Each thread mmaps a part of a large region of memory, then all threads access all of the pages in the large region in a random order. This simulates a widely shared region such as a memory-mapped library or a shared data structure like a hash table. In our microbenchmark, each thread maps a 64 KB region.

These benchmarks capture common sharing patterns we have observed in multi-threaded applications.

We run the three microbenchmarks and Metis on three different virtual memory systems: Linux (kernel version 3.5.7 from Ubuntu Quantal), Linux with the Bonsai VM system (based on kernel version 2.6.37), and RadixVM on our research OS. The microbenchmarks and Metis compile and run on both Linux and our OS without modifications.

All experiments are performed on an 80 core machine with eight 2.4 GHz 10 core Intel E7-8870 chips and 256 GB of RAM on a Supermicro X8OBN base board. Each core has 32 KB of L1 data cache and 256 KB of L2 cache and each chip has a shared 30 MB L3 cache. Experiments that vary the number of cores enable whole chips at a time in order to keep each core's share of the L3 caches constant. We also report single core numbers for comparison. For both application benchmarks and microbenchmarks, we take the average of three runs, though variance is consistently under 5% and typically well under 1%.

5.2 Metis

There are two factors that determine the scalability of Metis: contention induced by concurrent mmaps during the Map phase and contention induced by concurrent pagefaults during the Reduce phase. If the memory allocator uses a large allocation unit, Metis can avoid the first source of contention because the number of mmap invocations is small. Therefore, we measure Metis using two different allocation units: 8 MB to stress pagefault and 64 KB to stress mmap. In the 8 MB configuration, Metis invokes mmap 5,987 times at 80 cores, and in the 64 KB configuration, it invokes mmap 572,292 times. In both cases, it invokes pagefault approximately 12 million times, where 70% of these page faults cause it to allocate new physical pages and the rest bring pages already faulted on another core in to the per-core page tables.

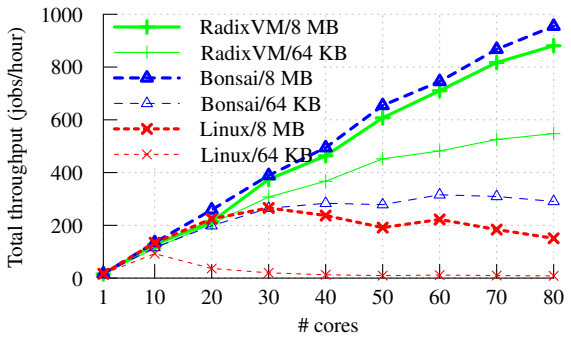

Figure 4: Metis scalability for different VM systems and allocation block sizes.

Figure 4 shows how Metis scales for the inverse indexing application for the three VM systems. Metis on RadixVM scales well with both large and small allocation sizes, substantially out-performing and out-scaling Linux in both configurations, and out-performing Bonsai in the 64 KB mmap-heavy configuration. For the 8 MB pagefault-heavy configuration, RadixVM and Bonsai scale equally, though the sequential performance of RadixVM's overall implementation is not as optimized as Linux, so RadixVM is $\sim 5\%$ slower than Bonsai at all core counts.

On Linux, both concurrent mmaps and concurrent pagefaults contend for the address space lock and, as a result, Metis on Linux scales poorly with both small and large allocation units. For large allocation units, pagefaults from different cores contend for read access to the read/write lock protecting the address space. For small allocation units, mmaps from different cores contend for write access to this same lock.

Metis on Bonsai scales well with large allocation sizes, because, much like in RadixVM, pagefaults in the Bonsai VM don't acquire contended locks or write to shared cache lines and hence run concurrently and without cache line contention. The fact that RadixVM's performance is within $\sim 5\%$ of Bonsai's performance indicates that there is little or no performance penalty to RadixVM's very different design; it's likely we could close this gap with further work on the sequential performance of our OS. With small allocation sizes, Bonsai doesn't scale well because of lock contention between mmaps.

5.3 Microbenchmarks

To better understand the behavior of RadixVM under the different address space usage patterns exhibited by Metis and other applications, we turn to microbenchmarks.

Figure 5 shows the throughput of our three microbenchmarks on RadixVM, Bonsai, and Linux. For consistency, we measure the total number of pages written per second in all three benchmarks. In RadixVM, because of per-core page tables, each of these writes translates into a page fault, even if the page has already been allocated by another core. Because Linux and Bonsai use shared page tables, they incur fewer page faults than RadixVM on the pipeline and global microbenchmarks.

The local microbenchmark scales linearly on RadixVM. This is the ideal situation for RadixVM because each thread uses a separate part of the address space. Indeed, the benchmark causes essentially zero cross-socket memory references. We observe about 75 L2 cache misses per iteration owing to our CPU's small L2 cache (about 64 of these are from page zeroing) and about 50 L3 cache misses, also mostly from page zeroing, but all are satisfied from the core's local DRAM. Likewise, because RadixVM can track remote TLBs precisely, the local microbenchmark sends no TLB shootdowns. Because there is no lock contention, a small and fixed number of cache misses, no remote DRAM accesses, and no TLB shootdowns, the time required to mmap, pagefault, and munmap is constant regardless of the number of cores. Linux and Bonsai, on the other hand, slow down as we add more cores. This is not unexpected: Linux acquires the shared address space lock three times per iteration and Bonsai twice per iteration, effectively serializing the benchmark. Furthermore, RadixVM has good sequential performance: at one core, RadixVM's performance is within 8% of Linux, and it is likely this can be improved.

The pipeline microbenchmark scales well on RadixVM, though not linearly, and significantly outperforms Linux and Bonsai above one core. We observe similar cache miss rates as the local microbenchmark and cross-socket memory references consist solely of pipeline synchronization, synchronization to return freed pages to their home nodes when they are passed between sockets, and cross-socket shootdown IPIs. As expected, every munmap results in exactly one remote TLB shootdown. However, the pipeline benchmark falls short of linear scalability because this one TLB shootdown takes longer on average as we add cores; the protocol used by the APIC hardware to transmit the inter-processor interrupts used for remote TLB shootdowns appears to be non-scalable. Linux and Bonsai behave much like they did for the local microbenchmark and for the same reasons and, again, RadixVM's single core performance is within 8% of Linux.

Finally, the global microbenchmark also scales well on RadixVM, despite being conceptually poorly suited to RadixVM's per-core page tables and targeted TLB shootdown. In this benchmark, RadixVM's performance is limited by the cost of TLB shootdowns: at 80 cores, delivering shootdown IPIs to the other 79 cores and waiting for acknowledgments takes nearly a millisecond. However, at 80 cores, the shared region is 20 MB, so this cost is amortized over a large number of page faults. Linux and Bonsai do better on this benchmark than on the local and pipeline benchmarks because it has a higher ratio of page faults to mmap and munmap calls. Furthermore, because Linux and Bonsai use a single page table per address space, they incur only a single page fault per mapped page, while RadixVM incurs n page faults per

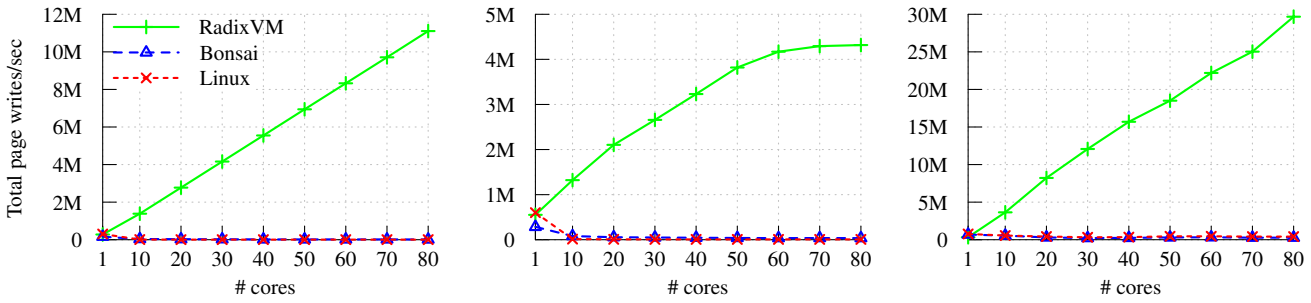

Figure 5: Throughput of local, pipeline, and global microbenchmarks as the total number of pages written per second.

mapped page for n cores. Ultimately, this has little effect on performance: the majority of these page faults only fill a page table entry without allocating a new backing page; even at 80 cores, when the locks in the radix tree are heavily contended, these "fill" faults take only 1,200 cycles.

5.4 Memory overhead

One potential downside of RadixVM is increased memory overhead both because a radix tree is a less-compact representation of virtual memory metadata than a traditional binary tree of memory regions and because page tables are per-core instead of shared.

To quantify the memory overhead of the radix tree, we took snapshots of the virtual memory state of various memory-intensive applications and servers running on Linux and measured the space required to represent the address space metadata in both Linux and RadixVM. Linux uses a single object to represent each contiguous range of mapped memory (a "VMA"), arranged in a red-black tree, which makes its basic metadata about address space layout very compact. As a result, Linux must store information about the physical pages backing mapped virtual pages separately, which it cleverly does in the hardware page table itself, making the hardware page table a crucial part of the address space metadata. RadixVM, on the other hand, stores both OS-specific metadata and physical page mappings together in the radix tree and can freely discard and reconstruct hardware page tables.

Table 2 summarizes the memory overhead of these two representations for four applications: the Firefox and Chrome web browsers, after significant use by the authors, and the Apache web server and MySQL database server used by the EuroSys 2013 paper submission web site. Considered in the context of the memory actually used by each application, the radix tree constituted at most 3.7% of the application's total memory footprint (RSS).

While RadixVM can freely discard hardware page tables under memory pressure, it's interesting to consider the memory overhead of per-core page tables relative to shared page tables. Per-core page tables could potentially require n times more memory than shared page tables for n cores; however, we expect most applications to exhibit better partitioning than this. For example, Metis allocates about 38 GB of memory

		Linux		Radix tree
	RSS	VMA tree	Page table	(rel. to Linux)
Firefox	352 MB	117 KB	1.5 MB	3.9 MB (2.4×)
Chrome	152 MB	124 KB	1.1 MB	2.4 MB (2.0×)
Apache	16 MB	44 KB	368 KB	616 KB (1.5×)
MySQL	84 MB	18 KB	348 KB	980 KB (2.7×)

Table 2: Memory usage for alternate VM representations.

when running on 80 cores; a shared page table requires another 100 MB of memory (0.3% of the application's memory use) while per-core page tables require 1.3 GB (3.6% of the application's memory use). Per-core page tables incur 13× the memory overhead of a shared page table for Metis at 80 cores, noticeably less than the worst-case 80×. While per-core page tables do require more memory, we believe that the overhead for most applications will amount to a small fraction of the overall memory use.

5.5 Breakdown of ideas

This section shows that each of the individual techniques employed by RadixVM is important to its scalability.

Radix tree. RadixVM uses a radix tree because it never induces cache line movement between non-overlapping accesses except during initialization, after collapsing, or when cache line granularity results in false sharing. To assess the scalability advantages of radix trees, we compare against a concurrent skip list, another data structure that could support a concurrent VM system. Our skip list implementation supports wait-free lookups and lock-free insert and delete operations [16]. An earlier design of RadixVM used this implementation until we discovered that it was a bottleneck.

To compare the skip list and the radix tree, we simulate an address space with 1,000 regions (a typical number for a large application). Readers continuously lookup a random key that is present (like a pagefault) while zero or more cores modify the data structure by continuously inserting a random key that is not present (corresponding to mmap) and then deleting that key (munmap).

Figure 6 shows the results for the skip list with 0, 1, and 5 writers. With only read sharing, lookup scales perfectly and

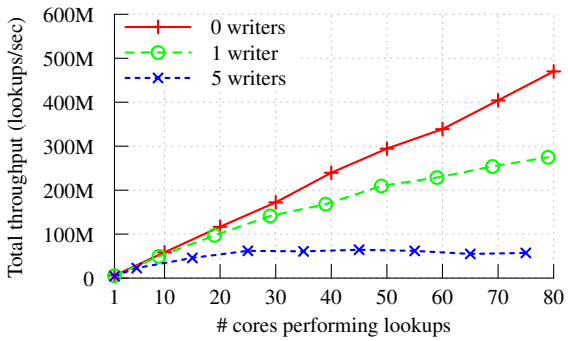

Figure 6: Total throughput of concurrent skip list lookups contending with inserts and deletes.

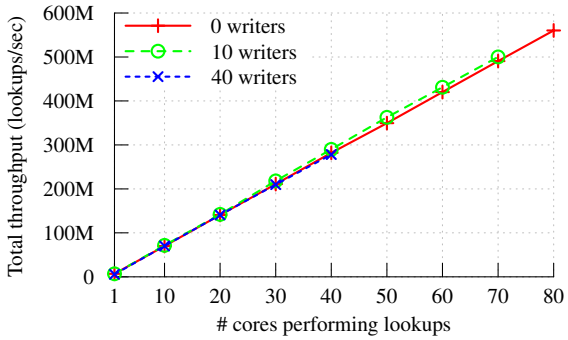

Figure 7: Total throughput of concurrent radix tree lookups contending with inserts and deletes.

each core adds ∼5.7 million lookups per second. This is to be expected because each reader has the complete skip list in its local cache and lookups don't induce cache coherence traffic.

However, with even 1 core writing and the remaining cores reading, lookups do not scale linearly. Even though the readers and writers use different keys and hence modify different regions of the skip list, lookup will occasionally read an interior node of the skip list that an unrelated insert modified to maintain the structure's balance, resulting in cache line contention that increases with the number of cores accessing the skip list. Adding more writers increases the likelihood of contention and with a mere five writers, lookup performance plateaus at 64 million lookups per second. Write performance also drops as the number of readers increases: a single writer can perform about 1.4 million insert/delete pairs per second with one concurrent reader, but only 0.3 million with 79 concurrent readers.

Figure 7 shows the equivalent results for the radix tree, this time with 0, 10, and 40 writers. In contrast with the skip list, the radix tree eliminates cache line contention by design: since there are no writes to internal nodes once they've been initialized, operations on unrelated keys don't result in cache line transfers. As a result, lookup throughput is unaffected by writers. Likewise, while insert/delete throughput is lower than the skip list (unlike a real address space, this benchmark has no locality, so nearly every insert allocates a 4 KB node), the insert/delete throughput does not change with the number of lookups.

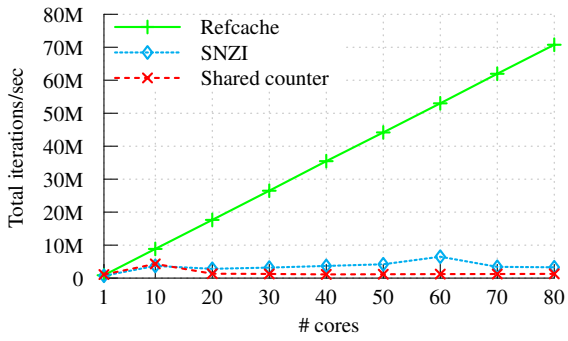

Figure 8: Page sharing throughput for different reference counting methods. Each iteration mmaps a shared physical page and then munmaps it.

Reference counting. To measure the importance of Refcache in isolation we use a microbenchmark that simulates the behavior of mapping and unmapping shared libraries into an address space. The microbenchmark allocates one page of memory, which n threads repeatedly mmap into the address space and then munmap, incrementing and decrementing the reference count of the underlying physical page constantly and concurrently from n cores. We measure this benchmark with three different versions of RadixVM: one with Refcache, one with a shared reference counter which is incremented and decremented with an atomic instruction, and one with an implementation of scalable reference counters based on SNZI [12].

Figure 8 shows the results of this comparison. The Refcache counters scale perfectly with an increasing number of cores, while the shared counter doesn't scale, as expected. SNZI counters substantially outperform the shared counter, but reach a scalability bottleneck as early as 10 cores. The authors of SNZI counters report better scalability on a Sun multicore computer [12], but our machine is much faster and achieves much higher absolute throughput even at 10 cores than the authors of SNZI report at 48 cores.

Refcache is able to achieve linear scalability in part because it delays zero detection, a trade-off that is worthwhile in RadixVM. While Refcache guarantees a zero reference count will be detected within two epochs, both shared reference counters and SNZI counters detect a zero reference count immediately, and hence require significantly more communication than Refcache.

Targeted TLB shootdown. Finally, we examine the benefits of RadixVM's per-core page tables with targeted TLB shootdowns versus shared page tables with broadcast shootdowns. Figure 9 shows the throughput of the three microbenchmarks using these two approaches. Unsurprisingly,

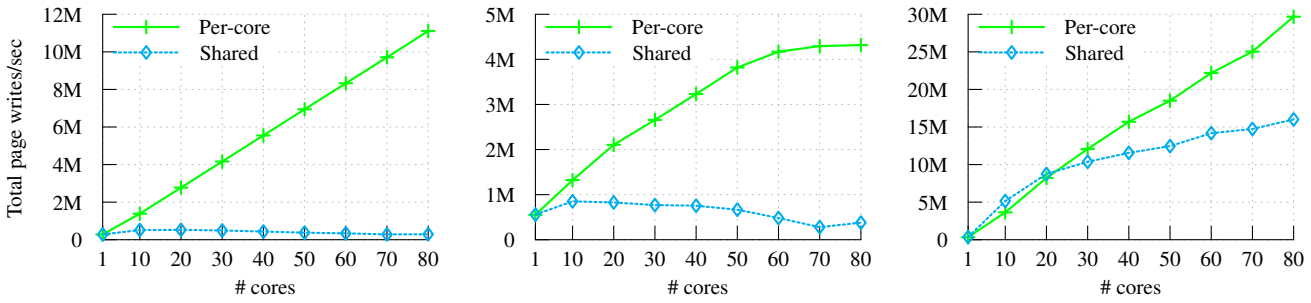

Figure 9: Throughput of local, pipeline, and global microbenchmarks using per-core page tables and shared page tables.

the local and pipeline benchmarks suffer tremendously. Both benchmarks munmap individual pages on every core in every iteration, and without the ability to track usage of these pages, they must broadcast TLB shootdowns to all cores executing the benchmark in every iteration. At ∼500,000 cycles per shootdown, both benchmarks grind to a halt. The performance of the two schemes for the global microbenchmark is much closer because this benchmark requires TLB shootdowns to all cores executing the benchmark under both schemes. However, despite the fact that, for n threads, per-core page tables require this benchmark to take n times as many page faults as shared page tables, per-core page tables nevertheless outperform shared page tables because they eliminate cache contention for accessing and modifying the shared page table structure.

5.6 Discussion

The results in this section show that RadixVM achieves its goal of allowing VM operations on non-overlapping regions to scale with the number cores and that the combination of techniques employed by RadixVM is necessary to achieve this. RadixVM's sequential performance is within $\sim 5\%$ of Linux, even though RadixVM has not been optimized for single-core performance. As expected, RadixVM requires more memory for address space structures than Linux: address space metadata requires $\sim 2\times$ more memory and per-core page tables have potentially much higher overhead (Metis exhibited $13\times$ overhead); however, both represent a small fraction of total application memory use. Furthermore, there are several techniques that can mitigate per-core page table overhead, including simply discarding page table memory.

6 Conclusion

This paper presented RadixVM, a new virtual memory design that allows VM-intensive multithreaded applications to scale with the number of cores. To achieve scalability, RadixVM avoids cache line contention using three techniques: radix trees, Refcache, and targeted TLB shootdowns. An evaluation using the Metis MapReduce library and microbenchmarks that capture a range of common virtual memory use patterns shows that RadixVM achieves its goal. We hope RadixVM's design will inspire other OS developers to provide scalable virtual memory primitives that obviate the need for application-level workarounds.

The source code to RadixVM is available at `http://pdos.csail.mit.edu/multicore`.

Acknowledgments

We thank Silas Boyd-Wickizer, Yandong Mao, Xi Wang, the anonymous reviewers, and our shepherd, Miguel Castro, for their feedback. This work was supported by Quanta Computer, by Google, and by NSF awards 0915164 and 0964106.

References

[1] FreeBSD source code. `http://www.freebsd.org/`.

[2] J. Appavoo, D. da Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems*, 25(3), Aug. 2007.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[4] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation lookaside buffer consistency: a software approach. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 113–122, Boston, MA, Apr. 1989.

[5] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[6] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[7] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Concurrent address spaces using RCU balanced trees. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, Mar. 2012.

[8] J. Corbet. The search for fast, scalable counters, May 2010. `http://lwn.net/Articles/170003/`.

[9] R. Cox, M. F. Kaashoek, and R. Morris. Xv6, a simple Unix-like teaching operating system. `http://pdos.csail.mit.edu/6.828/2012/xv6.html`.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] J. DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Nov. 1990.

[12] F. Ellen, Y. Lev, V. Luchango, and M. Moir. SNZI: Scalable nonzero indicators. In *Proceedings of the 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Portland, OR, Aug. 2007.

[13] J. Evans. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference*, Ottawa, Canada, Apr. 2006.

[14] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 87–100, New Orleans, LA, Feb. 1999.

[15] S. Ghemawat. TCMalloc: Thread-caching malloc, 2007. `http://gperftools.googlecode.com/svn/trunk/doc/tcmalloc.html`.

[16] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[17] P. W. Howard and J. Walpole. Relativistic red-black trees. `http://web.cecs.pdx.edu/~walpole/papers/ccpe2011.pdf`.

[18] P. W. Howard and J. Walpole. Relativistic red-black trees. Technical Report 10-06, Portland State University, Computer Science Department, 2010.

[19] ISO. *ISO/IEC 14882:2011(E): Information technology – Programming languages – C++*. ISO, Geneva, Switzerland, 2011.

[20] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *Proceedings of the ACM EuroSys Conference*, Leuven, Belgium, Apr. 2006.

[21] R. Liu and H. Chen. SSMalloc: A low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the 3rd Asia-Pacific Workshop on Systems*, Seoul, South Korea, July 2012.

[22] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, May 2010.

[23] P. McKenney. Hierarchical RCU, Nov. 2008. `https://lwn.net/Articles/305782/`.

[24] Microsoft Corp. Windows research kernel. `http://www.microsoft.com/resources/sharedsource/windowsacademic/researchkernelkit.mspx`.

[25] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 2006 ACM SIGPLAN International Symposium on Memory Management*, Ottawa, Canada, June 2006.

[26] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. *SIGPLAN Notices*, 46 (11):79–88, June 2011.

[27] L. Torvalds et al. Linux source code. `http://www.kernel.org/`.

[28] V. Uhlig. The mechanics of in-kernel synchronization for a scalable microkernel. *ACM SIGOPS Operating Systems Review*, 41(4):49–58, July 2007.

[29] L. Wang. Windows 7 memory management, Nov. 2009. `http://download.microsoft.com/download/7/E/7/7E7662CF-CBEA-470B-A97E-CE7CE0D98DC2/mmwin7.pptx`.

[30] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.