Scalable Address Spaces Using RCU Balanced Trees

Austin T. Clements M. Frans Kaashoek Nickolai Zeldovich
MIT CSAIL
{aclements,kaashoek,nickolai}@csail.mit.edu

Abstract

Software developers commonly exploit multicore processors by building multithreaded software in which all threads of an application share a single address space. This shared address space has a cost: kernel virtual memory operations such as handling soft page faults, growing the address space, mapping files, etc. can limit the scalability of these applications. In widely-used operating systems, all of these operations are synchronized by a single per-process lock. This paper contributes a new design for increasing the concurrency of kernel operations on a shared address space by exploiting read-copy-update (RCU) so that soft page faults can both run in parallel with operations that mutate the same address space and avoid contending with other page faults on shared cache lines. To enable such parallelism, this paper also introduces an RCU-based binary balanced tree for storing memory mappings. An experimental evaluation using three multithreaded applications shows performance improvements on 80 cores ranging from 1.7× to 3.4× for an implementation of this design in the Linux 2.6.37 kernel. The RCU-based binary tree enables soft page faults to run at a constant cost with an increasing number of cores, suggesting that the design will scale well beyond 80 cores.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management; E.1 [*Data Structures*]: Trees; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Algorithms, Design, Performance

Keywords RCU, Virtual memory, Multicore, Lock-free algorithms, Concurrent balanced trees, Scalability

1. Introduction

A common parallel programming model is shared-memory multithreading, where all threads of an application share a single address space. Such shared address spaces require the kernel virtual memory (VM) system to be likewise concurrent, and the kernel's approach to concurrent address space operations can severely affect the concurrency of a multithreaded VM-intensive application as a whole. In fact, it is not uncommon for applications to use processes instead of threads to avoid a single, shared address space, but this complicates sharing. Applications can also unknowingly make intensive use of VM: libc may call mmap or munmap internally to grow the address space or memory-map files, and even a seemingly innocent memory access can result in a soft page fault that modifies the application's shared page tables. The overall goal of this paper is to make such multithreaded applications that use virtual memory intensively (knowingly or unknowingly) scale well.

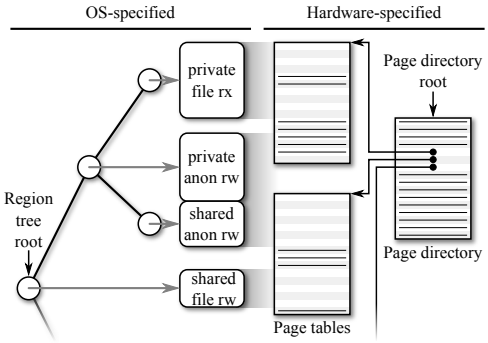


Figure 1. An address space with four mapped regions. Crossed-out page table entries do not map pages. Some pages have not been soft faulted yet, and thus do not map pages, despite being in a mapped region.

In most widely-used operating systems, an address space consists principally of a set of memory mapping regions and a page table tree, as shown in Figure 1. Each memory mapping region describes a range of virtual addresses and stores information about the mapping such as protection bits and backing files. Most widely-used operating systems use a tree to store the memory regions because applications often have thousands of memory regions (e.g., due to dynamic linking) and a tree enables the operating system to find the region containing a particular virtual address quickly. The page tables record the architecture-specific mapping from virtual pages to physical pages and, unlike the region tree, their structure is dictated by the hardware. This paper assumes x86-64 four-level page tables for concreteness, but the ideas are not specific to the x86.

Operating systems provide three address space operations of particular interest: the mmap and munmap system calls (collectively referred to as "memory mapping operations"), and page faults, which happen when the hardware cannot map a virtual address to a physical address. mmap creates memory mapping regions and adds them to the region tree. munmap removes regions from the tree and invalidates entries in the hardware page table structures. Page faults look up the faulting virtual address in the region tree and, if the virtual address is mapped, install an entry in the hardware page table and resume the application. We specifically focus on "soft" page faults, which may allocate new pages and new page tables or map pages already present in memory, but which do not page in data from an external drive.

The region tree and hardware page tables are shared by all threads in a process. To ensure correct behavior when several cores

ASPLOS'12, March 3–7, 2012, London, England, UK.

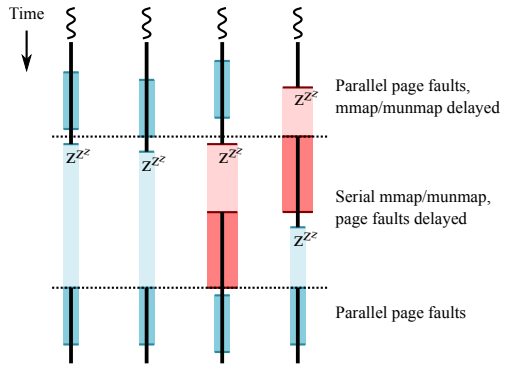


Figure 2. Linux address space concurrency with four threads. Page faults (skinny blue rectangles) and memory mapping operations (wide red rectangles) delay each other. Threads are sleeping in regions marked Z^{Z^Z}.

perform mmap, munmap, and page faults, most operating systems use several locks. Widely-used systems such as Solaris and Linux use a single read/write lock per process, which serializes page faults and memory mapping operations as shown in Figure 2, limiting the scalability of multithreaded VM-intensive applications. In these operating systems, a process can perform only one memory mapping operation at a time, and these operations also delay page faults. Page faults for different virtual addresses run concurrently, but block other threads from performing memory mapping operations.

Our goal is to make page faults scale to large numbers of cores. This requires addressing two basic forms of serialization. First, we must allow soft page faults to run concurrently with mmap and munmap operations to eliminate serialization on the per-process read/write lock. This is difficult because all three operations modify the address space data structures (the region tree, the page tables, or both). More subtly, we must also make page faults run without modifying shared cache lines to eliminate serialization caused by processor caches acquiring the cache line in an exclusive state. This is difficult because locks, even in read mode, require modifying shared cache lines.

To achieve our goal, this paper presents a new concurrent address space design that eliminates the above sources of contention by applying read-copy-update (RCU) [15] to the VM system and by introducing the BONSAI tree, an RCU-based concurrent balanced tree used as part of the design. While RCU is widely adopted in the Linux kernel, it has not been applied to the kernel's address space structures because of two significant challenges: the address space structures obey complex invariants that make RCU's restrictions on readers and writers onerous, and fully applying RCU to the address space structures requires an RCU-compatible concurrent balanced tree, for which no simple, applicable solutions exist.

To evaluate our approach, we implemented our concurrent address space design in Linux 2.6.37. This design leverages the BONSAI tree to apply RCU pervasively to both page table and tree operations, which allows page faults on distinct virtual addresses to proceed with no contention due to either locks or exclusive cache line accesses. For three VM-intensive workloads—Metis and Psearchy, from MOSBENCH [4], and Dedup from Parsec [2]—our design improves throughput on 80 cores by 3.4×, 1.8×, and 1.7× respectively, relative to stock Linux. Further, a microbenchmark demonstrates that the cost of handling a page fault in our design is nearly constant, independent of concurrent page faults or memory mapping operations, while this same cost grows rapidly in stock Linux as page fault and memory mapping load increases.

This paper contributes a novel design for an RCU-compatible balanced tree (BONSAI), a new address space design that uses BONSAI to achieve highly-scalable page fault handling, an implementation of that design in Linux, and an experimental evaluation using applications and microbenchmarks on an 80 core machine. Although our implementation is for Linux, we believe that our design could also be applied to other operating systems; while the details of address space management differ between kernels, most shared-memory kernels have similar address space representations.

The rest of the paper is organized as follows. Section 2 relates our contributions to previous work. Section 3 describes the design of the BONSAI tree. Section 4 covers existing address space designs in depth, using Linux as a case study. Section 5 describes our design for concurrent address space operations and section 6 summarizes its implementation in Linux. Section 7 reports on the experimental evaluation of our design and how it compares to the stock Linux approach and section 8 provides a brief discussion of our findings. Finally, section 9 summarizes our conclusions.

2. Background and related work

Address spaces in modern operating systems. Address spaces can consist of a large number of virtual address regions. As one example, on a typical Linux desktop, GNOME applications and web browsers such as Firefox and Chrome use nearly 1,000 distinct memory regions in each process. To manage this large number of regions, most modern operating systems use structures like the ones in Figure 1 to represent an address space. Linux uses a red-black tree for the regions, FreeBSD uses a splay tree, and Solaris and Windows (prior to Windows 7) use AVL trees [18, 24]. Linux and Solaris use a single read/write lock per address space. FreeBSD is more fine-grained: it uses a lock per region and a single write lock for mutating operations on an address space, serializing soft page faults with address space changes. Prior to Windows 7, Windows used a system-wide *PFN lock* [18]. Windows 7 uses fine-grained locking with better reported scaling results [24], but exact details are not public.

Since soft page faults must still modify address space structures (e.g., an entry in a page table), fine-grained locks are typically used in conjunction with per-process read locks to protect simultaneous page faults to nearby virtual addresses. The design proposed in this paper eliminates the read locks for page faults, but maintains the fine-grained locks for updates to page table entries. In many applications, including the three applications we study in this paper, these fine-grained locks are uncontended and typically stay in a core's cache, thus incurring negligible overhead.

Workarounds. It is not uncommon for system programmers to change their applications to use processes instead of threads to avoid contention caused by shared address space locking. For example, Psearchy, from MOSBENCH [4], was modified to use processes instead of threads. However, it is difficult to migrate an application from a shared address space to processes because sharing is less straightforward, and the added complexity makes it harder to maintain and evolve the application. Furthermore, using processes instead of threads can incur a performance penalty, since every process must fault on every shared page, rather than the entire application taking at most one fault per page [3].

Super pages can also increase scalability. On x86-64, this reduces the number of page faults by a factor of 512 (other platforms are similar), substantially reducing contention on the address space lock. However, applications that map many small regions (files or anonymous memory) cannot benefit from this solution. Furthermore, as memory demands and core counts grow, super pages will encounter the same scalability problems that regular pages currently contend with.

This paper presents a design that makes shared address space operations scalable, so that multithreaded applications can scale naturally to large numbers of cores, without the need to work around kernel limitations.

RCU. Our approach to increasing concurrency for address-space operations exploits the two key ideas of RCU [13]: allowing read operations to proceed without read locks, and delayed freeing of data structures to avoid races where a writer frees data while a reader is still using it. The benefit of avoiding read locks is twofold: not only does it allow read operations to proceed during write operations, but it eliminates the overhead of acquiring a read lock even when there are no writers. Even in read mode, acquiring a lock requires fetching the lock's cache line from the core that last updated the lock status; on today's multicore processors a contended cache line can take hundreds of cycles to fetch from a remote core, and RCU avoids this cost entirely.

Write operations, which coordinate using an exclusive lock, may change a data structure and remove entries from it while read operations are still using the removed entries. To ensure that these concurrent read operations do not observe invalid data, RCU delays freeing memory until it is guaranteed that all read operations that started before the data structure update have finished, at which point no further operations can observe the old data. Various schemes have been proposed to do the garbage collection of these delay-freed items, and the Linux kernel provides implementations ranging from a classic scheme that uses the scheduler to detect quiescence [17] to an epoch-based "sleepable" RCU implementation [14].

RCU data structures. The Linux kernel has increasingly adopted RCU for kernel data structures to improve scalability [16], but applying RCU is often not straightforward: RCU requires that mutating operations update only *one* pointer and that readers read each pointer at most once. This makes the update appear atomic: a reader will appear to execute either entirely before a concurrent update or entirely after, depending on when it reads the one updated pointer. More complex data structures such as balanced trees where mutation operations require multiple updates do not obviously fit with RCU, and none of the balanced trees in the Linux kernel use RCU. The BONSAI tree introduced by this paper is one solution to applying RCU to complicated data structures.

Trees with lock-free lookup. Fraser [6, 7] introduced a clever design for a red-black tree that allows read operations to proceed without locks, and write operations on different parts of the tree to proceed in parallel using fine-grained locks. To balance a tree, Fraser's design uses five fine-grained locks, one for each node involved in the balancing operation. Several other data structures, such as concurrent skip lists [20] and concurrent B-trees [10] also provide lock-free lookups with fine-grained write parallelism. However, such fine-grained parallelism is a poor match for address space concurrency because memory mapping operations must maintain invariants across *multiple* tree operations; maintaining internal tree consistency is not enough. The BONSAI tree takes advantage of the existing address space write-locking, and targets a simpler point in the design space that avoids the need for fine-grained locks.

Howard and Walpole [9] propose a relativistic red-black tree that supports lock-free lookups. By starting with a traditional lock-based tree, their design must explicitly handle tree rotations by considering several special cases. In contrast, BONSAI's design is based on an intuitively-correct tree design from functional languages, which achieves high performance with one key optimization to avoid path copying (described in §3.3). BONSAI's design is conceptually similar to Herlihy's approach for implementing concurrent data objects [8], but BONSAI derives additional simplicity by not handling concurrent modifications, which requires significant complexity in Herlihy's approach. All of the code for BONSAI, except for delete, is included in this paper, and the complete implementation is only 180 lines of code.

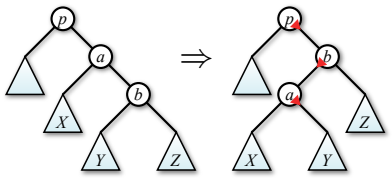


Figure 3. Single left rotation at a. Circles represent tree nodes, and triangles represent subtrees. The three updated pointers, at nodes p, a, and b are marked with red triangles.

3. An RCU-friendly balanced tree

This section describes the BONSAI tree, an RCU-compatible balanced binary tree based on the principle of non-destructive updates. As an RCU-compatible data structure, the BONSAI tree is designed to allow read operations such as lookup to proceed without locks in parallel with mutation operations such as insert and delete. Beyond the immediate concurrency implications of allowing read operations to proceed in parallel with writes, on large multicore machines, there are benefits to avoiding the cache coherence traffic caused by read locks. If the working set is in the local core's cache, read operations can proceed without any remote fetches. A single write lock is still used to coordinate mutation operations, but in many practical situations, including the Linux virtual memory system, the user of the tree must also ensure higher-level invariants across multiple tree operations and thus requires exclusive locks regardless.

The challenge to designing an RCU-compatible binary tree is keeping the tree balanced without introducing races with lock-free read operations, so that tree operations run in $O(\log n)$ time in the size of the tree. Maintaining balance requires mutation operations to perform tree *rotations*, which update multiple pointers across multiple tree nodes. If these updates are performed non-atomically, a concurrent, lock-free lookup may encounter a race condition where it observes only some of the pointer updates and returns an incorrect result.

For example, consider a single left rotation performed by an insert in order to rebalance a subtree—like the one pictured in Figure 3—and a concurrent lookup destined for a value in subtree Z. If the lookup reaches node a and then the rotation happens, it will proceed into subtree Y instead of subtree Z and miss the value it is searching for. The BONSAI tree avoids such races by design.

3.1 Approach

The design of the BONSAI tree is inspired by persistent data structures from functional programming. Such data structures are never modified in place; rather, the functional equivalent of a mutation operation leaves the existing data structure intact and returns a new instance of the data structure that reflects the change. This is kept efficient by sharing as much of the structure as possible between the original and "modified" instances.

Functional binary trees are an excellent example of this concept. Inserting into a functional binary tree constructs a new tree containing the original tree's elements, plus the newly inserted element. The two trees share all nodes except for those along the path from the newly inserted element to the root. Hence, like their in-place counterparts, insert into a functional tree requires time proportional to the height of the tree: it must traverse down the tree to find the insertion location, and then it must traverse back up the tree to construct the new path to the root and, ultimately, the new root node. Balanced trees are implemented by performing tree rotations as

```
typedef struct node
{
  struct node *left, *right;
  unsigned int size;
  int value;
} node_t;
```

Figure 4. Node structure

```
void
insert(node_t **root, int value)
{
  *root = doInsert(*root, value);
}

static node_t *
doInsert(node_t *node, int value)
{
  if (!node) return mkNode(NULL, NULL, value);

  if (value < node→value)
    return mkBalanced(node,
                      doInsert(node→left, value), node→right);
  if (value > node→value)
    return mkBalanced(node,
                      node→left, doInsert(node→right, value));
  return node;
}
```

Figure 5. Inserting into a tree

the tree is reconstructed from the bottom up, but, again, rotation is implemented by constructing a new, rotated subtree, rather than rotating a subtree in place.

Following this approach allows the BONSAI tree to be "correct by construction" in an RCU environment. In their simplest form, BONSAI operations construct a new tree non-destructively and then expose the modification in a single atomic step by updating a pointer to the tree's root node. If a lock-free reader reads the root node pointer before an update, it will operate entirely on the old tree; if it reads the root node pointer after an update, it will operate entirely on the new tree.

Steven Adams presents an elegant design for a functional sequential *bounded-balance tree* [1], and our tree derives from his design. Bounded-balance trees [19] exchange a certain degree of imbalance—controlled by a *weight* parameter—for fewer rotations than a strictly balanced tree. Combined with an optimization we describe in §3.3, this allows the BONSAI tree to produce fewer garbage nodes than a more conventional balanced tree, reducing stress on the memory allocator and CPU caches.

3.2 Algorithm

Figures 4 through 9 show the most important parts of an RCU-compatible bounded-balance tree, implemented in C but executing in a functional style. Each node has four fields (see Figure 4): a pointer to the left child, a pointer to the right child, the node's value, and a size field that records the number of nodes in the subtree rooted at that node.

doInsert, the core of insert, first recurses down the tree starting at the root until it falls off the edge of the tree. The recursive base case is easy: inserting a value into an empty tree simply creates a new node with no children. The actual work happens as doInsert unwinds its call stack, reconstructing the tree from the newly created node up to the root, ultimately returning a new tree that shares the majority of its structure with the original tree, but contains the

```
static node_t *
mkBalanced(node_t *cur, node_t *left, node_t *right)
{
  int ln = nodeSize(left);
  int rn = nodeSize(right);
  int value = cur→value;
  node_t *out;

  if (ln && rn && rn > WEIGHT * ln)
    out = mkBalancedL(left, right, value);
  else if (ln && rn && ln > WEIGHT * rn)
    out = mkBalancedR(left, right, value);
  else if (!UPDATE_IN_PLACE)
    out = mkNode(left, right, value);
  else
    goto updateInPlace;
  rcu_free(cur);
  return out;

updateInPlace:
  cur→left = left;
  cur→right = right;
  cur→size = 1 + ln + rn;
  return cur;
}
```

Figure 6. Balancing a subtree

inserted value. Finally, insert makes this entire updated tree visible to readers atomically by updating the pointer to the tree's root.

doInsert rebuilds each subtree using mkBalanced, shown in Figure 6. In its simplest form, mkBalanced takes a left subtree, left, and a right subtree, right, and returns a new node with value value conjoining the left and right subtrees. The reader can assume for now that UPDATE_IN_PLACE is false, as this enables an optimization which is described later.

If left and right subtrees are in balance (i.e., neither subtree contains more than a small multiple of the number of nodes in the other subtree), mkBalanced returns a new node with the given value and left and right children. If the new subtree would be out of balance, mkBalanced defers to either mkBalancedL or mkBalancedR, depending on which subtree is the smallest.

After constructing the new node, mkBalanced frees the original node being replaced, but does so in an RCU-delayed manner. Thus, the old node (and, in turn, the entire old tree) remains valid until every concurrent lookup that may be using the old tree completes.

mkBalancedL, shown in Figure 7, performs either single or double left rotation using singleL or doubleL, depending on the balance of the subtrees, returning the newly created, balanced subtree. singleL implements the single rotation of Figure 3, but does so without any in-place pointer updates, resulting in a new rotated subtree, like shown in Figure 8(a), where we see the two new nodes created by singleL, a' and b', as well as the new nodes created from p' up to the root by mkBalanced as doInsert unwinds its call stack. mkBalancedR, not shown, performs the equivalent right rotations.

lookup, shown in Figure 9, is identical to an ordinary, sequential tree look-up except for the enclosing calls to mark it as an RCU reader (which are no-ops in a non-preemptible Linux kernel). Because the BONSAI tree performs non-destructive updates and delay-frees all nodes, lookup does not require any locks, nor does it need any form of synchronization or atomic instructions on an x86 system.[1]

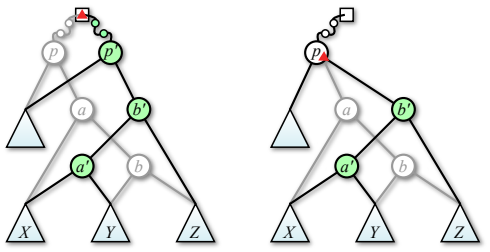
[1] Memory fence instructions are required on some non-x86 architectures, like the Alpha, for reading RCU-managed pointers between rcu_read_begin and rcu_read_end.

```
static node_t *
mkBalancedL(node_t *left, node_t *right, int value)
{
  if (nodeSize(right→left) < nodeSize(right→right))
    return singleL(left, right, value);
  return doubleL(left, right, value);
}

static node_t *
singleL(node_t *left, node_t *right, int value)
{
  node_t *out = mkNode(mkNode(left, right→left, value),
                       right→right,
                       right→value);
  rcu_free(right);
  return out;
}

static node_t *
doubleL(node_t *left, node_t *right, int value)
{
  node_t *out =
    mkNode(mkNode(left, right→left→left, value),
           mkNode(right→left→right, right→right,
                  right→value),
           right→left→value);
  rcu_free(right→left);
  rcu_free(right);
  return out;
}
```

Figure 7. Single and double left rotations



(a) Without optimization; O(log n) new nodes.

(b) With optimization; 2 new nodes.

Figure 8. Single left rotation at a, in functional style. The box represents the root node pointer. Garbage nodes are shown faded, new nodes are highlighted in green, and red triangles mark updated pointers.

```
bool
lookup(node_t **root, int value)
{
  rcu_read_begin();
  node_t *node = *root;
  while (node && node→value != value) {
    if (node→value > value)
      node = node→left;
    else
      node = node→right;
  }
  rcu_read_end();
  return node != NULL;
}
```

Figure 9. The lookup function

delete, not shown, has two cases. If the node to delete is a leaf, delete merely omits the node from the tree and rebuilds the path to the root using mkBalanced. If the node is an interior node, delete removes the node's successor from the tree (which must be a leaf and a descendant of the node being deleted) and substitutes it for the node being replaced, rebuilding from the successor up to the root.

3.3 Optimization

The implementation described so far allocates and frees $O(\log n)$ nodes for each insert. This is unavoidable in a true functional setting, where every version of the data structure must remain intact, but BONSAI has no such requirement. As presented, BONSAI delays the commit point—the time at which the inserted value becomes visible—until the final root pointer update. This is unnecessarily strict. For example, once a rotated subtree has been constructed, the remaining work is simply path copying to the root (assuming no additional rotations), but since this newly constructed path is structurally equivalent to the original path modulo the one pointer to the rotated subtree, we can atomically update this one pointer directly, rather than waiting to update the root pointer after reconstructing the entire path. The key to the resulting optimization is shown in Figure 8(b): once mkBalanced performs a non-destructive rotate, we can make that rotation visible immediately by updating a single pointer in p and avoid path copying. Furthermore, this optimization is safe under multiple rotations: since the contents of the tree are identical before and after a rotation, committing a rotation to the tree will have no affect on the results returned by a concurrent lookup.

By eliminating path copying, this optimization reduces the number of garbage nodes produced by insertion from $O(\log n)$ to $O(1)$ expected, which translates directly into less pressure on the memory allocator and the CPU cache, especially when combined with RCU delayed freeing. In practice, using a weight of 4, insertion performs only ∼0.35 rotations on average; with this optimization, this results in about 2 allocations and 1 free per insert on average, regardless of tree size.

The implementation of this optimization lies entirely in mkBalanced and does not affect the bottom-up insertion approach. If mkBalanced does not have to create a new node for rebalancing, it simply updates the current node in place. Typically, this will be a no-op, but if either left or right is the result of a rotation deeper in the tree, this will commit that rotation to the tree. mkBalanced also updates the size field in place; this need not be done atomically with the child pointer update because the size field is only accessed by writers.

Deletion can be optimized similarly, with one caveat. Since deleting an interior node requires two operations—removing the deleted node's successor and substituting it for the deleted node—delete must perform path copying from the successor up to the deleted node. By restricting itself to in-place updates only above the deleted node, delete accomplishes these two steps atomically.

This optimization is similar in effect to what Howard's relativistic red-black trees achieve [9], but its correctness is more readily apparent in BONSAI. Furthermore, it is generally applicable to other concurrent data structures derived from functional structures.

4. Address spaces in an operating system

Figure 1 in §1 briefly summarized the overall structure of an address space common to several kernels. An address space consists of a collection of non-overlapping memory regions or *virtual memory areas* (VMAs), where each area spans some set of pages in the virtual address space. On UNIX-like operating systems, these regions are manipulated by calls to memory mapping operations like mmap and munmap, though the regions may not correspond directly to the arguments to these calls. For example, an mmap adjacent to an existing VMA may simply extend that VMA, and an munmap in the middle of a VMA may split that VMA into two VMAs.

Each address space also maintains hardware page tables that map virtual addresses to physical addresses (on the x86, this consists of a tree of page directories and page tables, rooted at a root page directory). These hardware page tables are filled in lazily: creating a virtual memory area does *not* allocate pages or mark them present in the page tables. When an application tries to access a page of memory in a region spanned by some VMA for the first time, the hardware will generate a page fault, which the kernel will trap. The page fault handler looks for a VMA containing the faulting virtual address and, if one exists, only then will the kernel allocate a page and fill in the corresponding page table entry. Typically, the VMAs in an address space are kept in a balanced tree to facilitate efficiently looking up the VMA containing a faulting virtual address.

Multithreaded applications share an address space between threads, which, in most operating systems, means they share the set of VMAs as well as the hardware page tables. Memory mapping operations and page faults both access and modify these shared structures, and this concurrency must be addressed. Such address space designs suffer from four general classes of concurrency hazards, each mirroring some component of the address space structure: races involving the VMA tree, races involving VMA bound adjustment, races involving page table allocation and freeing, and races involving page allocation and freeing. These races are not simple coding errors, but design races that violate the semantics and invariants of address space operations. For example, in a naive solution, a race between an unmap operation and a page fault could result in a page being mapped in an otherwise unmapped region of memory.

The rest of this section uses the Linux kernel as a case study to illustrate how existing operating systems handle concurrent address space operations in multithreaded applications.

4.1 Linux case study: address space read/write locking

Linux, like many operating systems, protects address space structures using a per-process read/write lock, known as the mmap_sem read/write semaphore in Linux. This allows memory mapping operations like mmap and munmap, which acquire the lock in write mode, to perform complex operations atomically, while permitting page faults, which acquire the lock in read mode, to proceed in parallel with each other, as shown in Figure 2.

Running page faults concurrently is not straightforward, because page faults also modify the address space, and these modifications must be protected from concurrent page faults. At a minimum, a page fault must fill a page table entry, and it may have to allocate page tables or fill kernel-specific data structures. For this, Linux uses various fine-grained spinlocks, the two most notable of which are the page directory lock, which protects the insertion of new page directories and page tables into existing page directories, and the page table entry (PTE) lock, which protects the insertion of pages into page tables. Memory mapping operations generally do *not* acquire these fine-grained locks because they already acquire the mmap_sem in write mode, which prevents memory mapping operations and page faults from executing simultaneously. One exception is that munmap acquires PTE locks as it clears page tables. These locks are summarized below.

Lock	Type	Protects
mmap_sem	read/write per process	Address space
Page dir. lock	spinlock per process	Page dir. entries
PTE lock	spinlock per page table	Page table entries

Linux employs double-check locking to reduce both how frequently it must acquire these fine-grained locks and the duration they must be held for. For example, the page fault handler acquires the page directory lock only after observing a non-present page directory entry, and even then it acquires the lock only after optimistically allocating memory for the new page table. If a concurrent page fault has filled in the page directory entry in the meantime, it simply discards the optimistically allocated page table. Because of double-check locking, typically the only fine-grained lock the page fault handler acquires is the PTE lock. Indeed, recent versions of Linux have a separate PTE lock per page table to eliminate lock contention for all but nearby page faults (virtual addresses within 2 MB of each other on x86-64).

The write lock protects structures accessed and modified by memory mapping operations and the combination of the read lock and these fine-grained locks protects structures accessed and modified by the page fault handler, while still generally allowing page fault handlers to execute in parallel. Despite the scheme's sophistication, this locking has a concurrency cost: if any application thread is modifying the address space, no other thread can be taking page faults and if any thread is taking a page fault, no other thread can modify the address space. Furthermore, even in the absence of memory mapping operations, concurrent soft page faults still contend for the cache line that stores the state of the read/write lock. Under a heavy page fault load, the cost of this coherency traffic alone can affect application-level benchmark throughput.

5. Concurrent address spaces

In this section, we describe our concurrent address space design by introducing three increasingly-concurrent refinements of the baseline read/write locking employed by Linux. The first refinement, called fault locking, reduces the amount of time that memory mapping operations hold the mmap_sem read/write lock in write mode, allowing soft page faults to execute concurrently with the read-only parts of a memory mapping operation. The second refinement, called hybrid locking/RCU, eliminates the need for soft page faults to acquire the mmap_sem lock at all by applying RCU to the hardware page table structures and individual VMA entries, while protecting the VMA tree with a new read/write lock. The third refinement, called pure RCU, uses the RCU-based BONSAI tree to eliminate the read/write lock protecting the tree, eliminating any contended cache lines that must be modified by every soft page fault.

5.1 Fault locking

In this first refinement, we observe that memory mapping operations begin with a read-only phase in which they plan how to modify the address space before performing any data structure updates. In principle, page faults can safely proceed in parallel with the memory mapping operation during this time, hence permitting greater address space concurrency.

To take advantage of this observation, we add a *fault lock* to each address space and modify the page fault handler to acquire this lock in read mode, rather than the coarser mmap_sem. Memory mapping operations still acquire the mmap_sem in write mode and thus preclude races between each other, but before doing anything that could conflict with the page fault handler, they acquire the fault lock in write mode. The fault lock is released only when the mmap_sem is released, which, much like two-phase locking, keeps existing atomicity properties intact.

While memory mapping operations must acquire the fault lock before making any modifications, even before this, if they read anything that might be modified by a concurrent page fault, they must acquire the fault lock then to serialize the page fault. This complication will arise again in the RCU-based refinements, where it will require a more nuanced solution.

This refinement introduces no race conditions compared to stock Linux locking, but it only partially addresses the serialization between memory mapping operations and page fault handling, because it allows only marginally more concurrency between fault handlers and memory mapping operations.

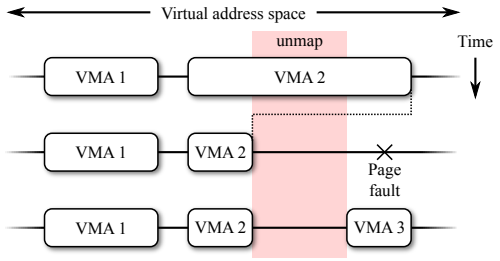


Figure 10. When splitting VMA 2 into two regions, the range originally covered by the top of VMA 2 can appear unmapped after VMA 2 is adjusted and before VMA 3 is created.

5.2 Hybrid locking/RCU

Our ultimate goal is to allow the page fault handler to run concurrently with both other page faults and memory mapping operations. Our solution to the problems that arise after eliminating the read lock around the page fault handler is twofold. Wherever possible, we apply RCU to eliminate the need for read locking. Then, in cases where a rare, non-benign race might arise, we detect inconsistencies and restart the page fault handler, this time with the mmap_sem held to ensure progress (similarly, we retry with locking in occasional "hard" cases).

The hybrid locking/RCU refinement approaches this goal without any structural changes to the address space. Other than the VMA tree, all of the data structures involved in the address space can be modified one step at a time without violating any invariants. This is not the case for the VMA tree, so this refinement still serializes access to the VMA tree using a read/write lock, which both the page fault handler and memory mapping operations acquire around tree operations. The pure RCU refinement, covered in the next section, tackles this last need for locking.

Removing the read lock around the page fault handler unleashes a number of races. We describe those that follow from the general address space design here and dive in to a few Linux-specific races in §6.

VMA split race. If no VMA contains the faulting address, the page fault could be a "segfault," or there may be a stack region that should be expanded to cover the faulting address, or we may be in the midst of a VMA split. The first two situations occur in stock Linux and we have to acquire the mmap_sem to handle them. The third situation is new and occurs because during a split, a region of memory can transiently appear unmapped, as shown in Figure 10. The munmap operation splits a VMA into two VMAs in two steps: it adjusts the bound of the existing VMA at time 2 and at time 3 it inserts the new VMA for the top part. Concurrently at time 2, a page fault handler may run that looks up an address in the top part of the VMA, which is momentarily not present. Since this race is unlikely, our approach is to simply retry the page fault with the mmap_sem held, which guarantees progress by preventing further races.

Page table deallocation race. Much like how pages are allocated on demand by soft page faults, page directories and tables are only allocated when a page fault encounters an empty page directory entry while installing mappings for a VMA that was faulted in. Conversely, unmapping a region recursively scans the page table tree, as illustrated in Figure 11, freeing not only the pages in that region, but also any page directories and tables that are no longer needed, clearing page directory entries that point to these. In the read/write locking refinement, the mmap_sem prevents a concurrent unmap and page fault within the same VMA. Without the mmap_sem, two problems arise from the race between page faults and the unmap scan. The first is classic RCU: the unmap may free VMAs, page

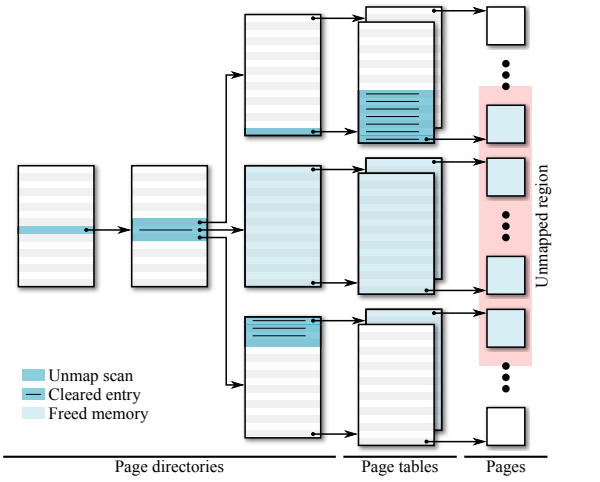


Figure 11. Unmapping a region of an x86-64 four-level page table tree. Unmapping recursively scans the page table tree to free unused page directories and page tables, in addition to freeing pages. Our design RCU-delays these frees. Unmapping must also clear page directory and page table entries that point to structures freed by the unmap, which introduces write-write races with page faults that fill these entries.

tables, or page directories that are being used by a concurrent page fault. To address this, we simply RCU-delay freeing these structures. The second is described next.

Page table fill race. A subtler race arises because both page faults and unmapping operations modify page directory and page table entries: page faults fill pointers to newly allocated pages and page tables, and unmapping operations clear pointers to freed page tables and pages. In both stock Linux and hybrid locking/RCU, fine-grained locking prevents simultaneous page fault handlers from filling the same page directory entry. However, this does not address the race between page faults and unmapping.

Consider a concurrent page fault and munmap on the same VMA, where the munmap has already surpassed the faulting address, delay freed the page table containing the faulting address, and cleared the corresponding page directory entry. The concurrent page fault will observe the blank page directory entry and could naively conclude that it needs to allocate a new page table and ultimately a new page. At best, these will never be freed; at worst, they could wreak havoc with a later mapping of the same region of memory. Luckily, this race is easy to detect: after acquiring the fine-grained PTE lock protecting the page table entry to be filled, but before actually filling it, the page fault handler double-checks that the VMA has not been marked as deleted and that the faulting address still falls within the VMA's bounds. In the unlikely event that the mapping of the faulting address has changed since the page fault's VMA lookup, this test will fail and the page fault handler will retry with the mmap_sem held. Curiously, this race check itself has a race: the mapping could still change after the check and before filling the page table entry, since mapping operations do not hold any fine-grained locks when manipulating VMAs. However, for numerous reasons, munmap does acquire PTE locks as it clears page tables, which is enough to ensure that, if the mapping does change, either the page fault handler's race check will observe the modified VMA, or munmap will observe and free pages allocated by the racing page fault.

While the solution to this race does not directly involve RCU, because of RCU, it is safe to *follow* page table pointers without any

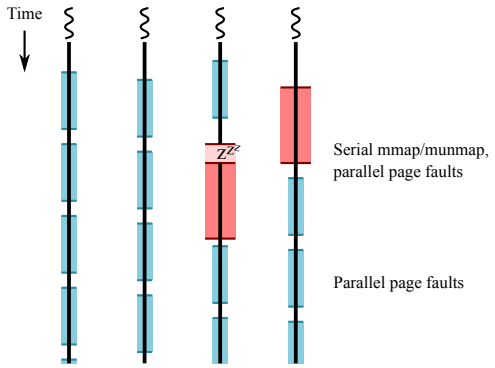


Figure 12. Pure RCU design concurrency with four threads. Page faults (skinny blue) neither delay memory mapping operations (wide red), nor are delayed by memory mapping operations, which benefits both page faults and memory mapping operations.

fine-grained locks. As a result, contention on the page directory lock and the PTE locks is virtually nonexistent.

Together, these three solutions address races involving bound adjustment and deletion of VMAs, allocation and freeing of page directories and page tables, and allocation and freeing of pages, three of the four major components of the address space. To address the fourth and final major component, the VMA tree, this refinement falls back on locking, which we remedy in our fourth concurrency refinement.

5.3 Pure RCU

Operating systems typically store the set of virtual memory areas in a balanced tree, such as a red-black or AVL tree, to facilitate fast lookup. However, these trees are not RCU-safe: as described in §3, a lock-less lookup during a concurrent insert or delete can result in the lookup failing, even if the element being looked-up is unrelated to the element being inserted or deleted.

Hence, to fully eliminate coarse-grained locking from the page fault handler, pure RCU further refines hybrid locking/RCU by replacing the locked red-black tree with the RCU-compatible BONSAI tree. This addresses races in all components of the address space, while BONSAI's lock-free lookups allow page faults to proceed fully in parallel both with each other and with memory mapping operations.

In contrast with the limited concurrency permitted by stock Linux's read/write locking, as shown in Figure 2, Figure 12 demonstrates the concurrency permitted by pure RCU page fault handling. In Figure 12, page fault handlers run concurrently with everything. As a result, the kernel can process page faults at a much higher rate and, furthermore, eliminating interference between page faults and memory mapping operations also improves memory mapping performance.

6. Implementation

We implemented the three new address space concurrency designs in the Linux 2.6.37 kernel. The implementation passes the Linux Test Project [11], as well as our own stress tests, and is further validated by a lock protection analysis we performed in the process of modifying the Linux virtual memory system and exhaustive schedule checking of a model of the VM system designed to capture key races. Our implementation consists of ∼2,600 lines of code, and the core BONSAI tree implementation is 180 lines of code.

The current implementation handles regular soft page faults in anonymous memory mappings. For memory-mapped files and copy-on-write faults, the implementation retries the page fault with the lock held. Fortunately, retrying is fast, so this decision poses little overhead: almost always, the VMA found on the first try will still be applicable on the second try (which can be checked easily), thus avoiding the entire VMA lookup, and the page directory walk will still be hot in the CPU cache.

The implementation was guided by a lock protection analysis we performed using a version of QEMU [21] modified to trace every memory access and provide a hypercall interface for recording memory allocations and reachability, lock acquisitions, and lock releases. Using type information from memory allocations, this analysis tool inspected how frequently different variables and structure fields were accessed with and without certain locks, thus inferring what data was likely to be protected by what locks and where in the source code those variables and fields were read and written. While this analysis was far from perfect, it was a valuable tool for understanding the Linux virtual memory system and finding potential concurrency hazards and races.

Several advanced features of the Linux virtual memory system complicated the RCU-based implementations and introduced additional potential races. Several of these races relate to Linux's *reverse map*, which tracks all virtual mappings of each physical page to facilitate operations such as swapping. For example, the page fault handler is responsible for associating an "anon_vma" with each anonymous memory VMA for tracking reverse mappings in that VMA. Luckily, this operation occurs infrequently, so the implementation simply retries the page fault in such cases. Likewise, the implementation handles various other uncommon operations—such as Linux's stack guard reservations—using the same retry-with-locking mechanism used when races are detected.

The mmap_cache, a seemingly innocent feature of the Linux page fault handler, causes surprising trouble. This caches the most recent VMA tree lookup in each address space. Any VMA lookup first checks if the cached VMA satisfies the request; if so, it can return immediately; if not, it looks up the VMA and stores the result in the cache. Maintaining this cache in an RCU environment, where a VMA can be doomed to deletion by the time it is looked up, is possible (albeit difficult), but with many threads faulting on different VMAs, it does more harm than good: the hit rate will be low (below 1% in our benchmarks), so every page fault will update it, which can result in significant cache coherence traffic. Thus, for the two RCU-based designs, we simply disable the mmap_cache because it harms overall performance. A more complete solution might dynamically disable the mmap_cache for multithreaded processes.

7. Evaluation

This section evaluates the following hypotheses:

- Can application scalability be limited by bottlenecks in the implementation of address spaces?
- Do the techniques proposed in this paper eliminate those bottlenecks?
- Is a simple fine-grained locking approach sufficient to remove the bottlenecks or is RCU necessary?
- Are short-duration read-locks acceptable in the soft page fault handler, or is the complete application of RCU necessary for scalability?
- Are application workarounds still necessary to achieve peak scalability?

7.1 Evaluation method

Only VM-intensive applications benefit from the designs described in this paper, so we selected three VM-intensive benchmarks: Metis [12], a single-server multicore implementation of MapReduce

from the MOSBENCH suite [4], used to compute a word position index from a 2 GB in-memory text file; Psearchy, a parallel text indexer based on searchy [23] (also from MOSBENCH), applied to the Linux 2.6.35-rc5 source tree; and Dedup, a deduplication-based file compressor from the Parsec benchmark suite [2], used to compress a 5.3 GB ISO image.

We selected these three benchmarks because they are multi-threaded (the other MOSBENCH applications are multi-process) and exhibit high page fault rates (the other Parsec applications do not), qualities we expect to see in more applications given trends towards higher core counts and larger data sets. Furthermore, the two MOSBENCH applications have alternate implementations that allow us to compare our general solutions with application-specific workarounds for address space scalability issues. The performance of other applications from the MOSBENCH and Parsec benchmark suites, which are not VM-intensive, is not affected by the three concurrent address space designs proposed in this paper.

We test the three benchmarks on Linux 2.6.37 with stock Linux as well as the three refinements described in §5. We also use microbenchmarks to evaluate the scalability and performance of just the page fault handler.

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. For both application benchmarks and microbenchmarks, we take the best of three runs to mitigate unrelated system activity. For application benchmarks, we spread enabled cores across sockets in order to maximizes the total cache and memory resources available to the application. The microbenchmarks are sensitive to cache size, so for these we group enabled cores on as few sockets as possible so that the per-core resources remain roughly constant across each experiment.

7.2 Application results

In all three applications, the greater address space concurrency of the pure RCU design results in significantly higher application throughput at large core counts; indeed, Metis and Dedup achieve near-perfect speed-up to 80 cores under the pure RCU design, while stock Linux read/write locking achieves 45× speed-up over single-core performance at best. Furthermore, as the number of cores increases, the gap between the pure RCU design and the three less-concurrent designs widens, suggesting that, as hardware becomes increasingly parallel, sophisticated concurrency schemes such as RCU and lock-free operation will become increasingly important.

Metis. Metis, shown in Figure 13, runs one worker thread per core, which together mmap and soft fault approximately 12 GB of anonymous memory as they generate intermediate tables between the map, reduce, and merge phases. As in the original MOSBENCH configuration, we use Metis with Streamflow [22], a scalable memory allocator, which mmaps allocation pools in 8 MB "segments," resulting in relatively few memory mapping operations compared to a more conservative allocator like glibc's malloc.

For Metis, the pure RCU design achieves near-perfect 75× speed-up at 80 cores and outperforms read/write locking by 3.4×. On the other hand, read/write locking's performance degrades with more cores because of both lock contention and cache contention for the mmap_sem. While the pure RCU design has less then 1% idle time at 80 cores, read/write locking has 12% idle time, waiting to acquire mmap_sem, and further spends 9.6% of its time contending for the mmap_sem's wait queue spinlock when it fails to acquire the lock. At the same time, it spends 31% of its time manipulating the mmap_sem cache line to acquire and release the lock. Finally, the increased contention in the kernel indirectly causes a 44% increase in the user time required to complete a job (e.g., due to increased cache pressure and interconnect traffic). Together, these factors account for the difference between the performance of read/write locking and

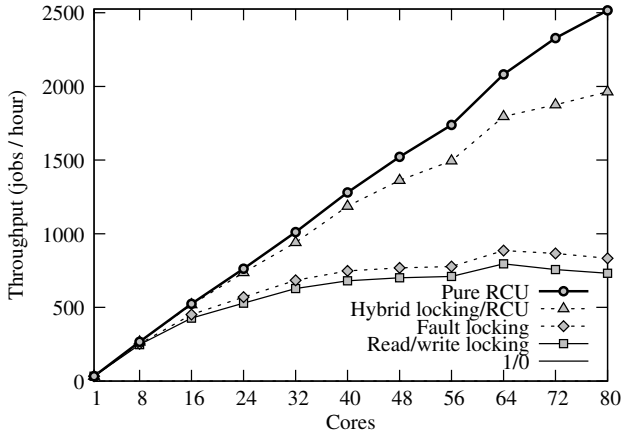


Figure 13. Metis throughput for each page fault concurrency design.

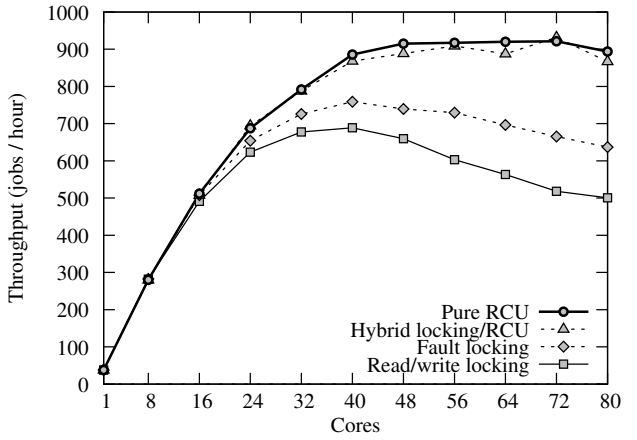


Figure 14. Psearchy throughput for each page fault concurrency design.

pure RCU at 80 cores. Fault locking shows little improvement over read/write locking. Hybrid locking/RCU outperforms read/write locking by 2.7×, but only achieves 59× speed-up because of the remaining contention on the lock protecting the VMA tree.

Psearchy. Psearchy is more heterogeneous than Metis, stressing both large and small memory regions. During initialization, Psearchy starts one worker thread per core, all of which simultaneously malloc and zero a 128 MB per-thread hash table; during main execution, these threads use libc's stdio library to read files, which internally calls mmap and munmap to manage memory for stream buffers, resulting in ∼30,000 small anonymous mmap/munmap pairs. As in the original MOSBENCH configuration, we index the Linux 2.6.35-rc5 source tree, consisting of 368 MB of text across 33,312 source files. To avoid IO bottlenecks, all input files are in the buffer cache and all output files are written to tmpfs.

The results for Psearchy are shown in Figure 14. Psearchy performs 13× more memory mapping operations per second than Metis, and, as a result of this high rate, the serialization of mmap and munmap ultimately bottlenecks Psearchy's performance in all four concurrency designs. With read/write locking, this base lock contention is further aggravated by page faults acquiring the mmap_sem in read mode; indeed, performance decays beyond the peak at 32 cores owing to the non-scalability of Linux's read/

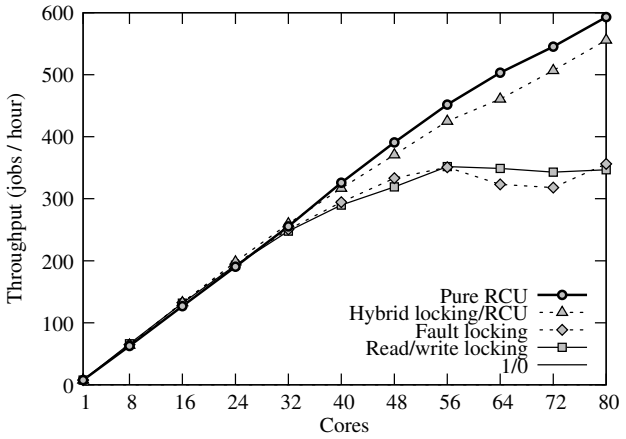


Figure 15. Dedup throughput for each page fault concurrency design.

write lock implementation. The other three designs perform better because they do not acquire the mmap_sem in the page fault handler, leaving only mmap and munmap to contend on that lock. Fault locking performs 27% better at 80 cores because it reduces this window of contention between memory mapping operations and page faults, even though it still acquires long-duration locks. Hybrid locking/RCU performs 1.7× better than stock because it nearly eliminates lock contention between memory mapping operations and page faults; its performance decays at high core counts because of contention for the cache line storing the lock that protects the VMA tree. Finally, the pure RCU design eliminates both lock and cache contention in the page fault handler, outperforming read/write locking by 1.8× and hybrid locking/RCU by 3.1%.

Metis and Psearchy provide alternate implementations specifically designed to work around address space scalability limits at the cost of increased application complexity. Metis can use 2 MB superpages, reducing the number of page faults by a factor of 512. Despite this optimization, unmodified Metis using the pure RCU design outperforms the optimized Metis using read/write locking; the former achieves 76× speed-up at 80 cores while the latter only 63× speed-up. Hence, for Metis, it is better to address the root problem in the kernel, rather than work around it in the application.

Psearchy requires relatively little shared state, so it can run in a multi-process configuration, avoiding shared address spaces altogether. Unlike Metis, Psearchy is ultimately limited both by lock contention between mapping operations and by lock contention within glibc itself, neither of which we address. Thus, while multi-threaded Psearchy using the pure RCU design significantly outperforms multi-threaded Psearchy using read/write locking, switching to a multi-process configuration proves even more effective, achieving 49× speed-up at 80 cores, versus 25× for multi-threaded Psearchy. Concurrent mapping operations would narrow this gap, but obviating this workaround requires improvements in user-space libraries as well as the kernel.

Dedup. Dedup, shown in Figure 15, deduplicates and compresses the blocks of an input file using a pipeline of communicating thread pools. We use a slightly modified version of Dedup that fixes one internal scalability bottleneck and substitutes jemalloc [5] for glibc's malloc because, in our experiments, jemalloc consistently outperformed and out-scaled glibc's malloc. Dedup's allocation pattern is similar to Metis: frequent heap allocation causes Dedup to mmap thousands of 4 MB–8 MB chunks and ultimately soft fault 13 GB of memory. As a result, the lock contention for the mmap_sem is comparatively low. For read/write locking and fault

Metis	user	sys	idle
Read/write locking	150 s	196 s	45 s
Fault locking	139 s	155 s	49 s
Hybrid locking/RCU	107 s	35 s	4 s
Pure RCU	102 s	11 s	1 s
Psearchy	user	sys	idle
Read/write locking	114 s	177 s	283 s
Fault locking	107 s	124 s	220 s
Hybrid locking/RCU	107 s	47 s	176 s
Pure RCU	107 s	12 s	201 s
Dedup	user	sys	idle
Read/write locking	465 s	146 s	248 s
Fault locking	567 s	35 s	239 s
Hybrid locking/RCU	426 s	18 s	93 s
Pure RCU	430 s	17 s	57 s

Table 1. Comparison of user, system, and idle time at 80 cores for a single "job" in each concurrency design.

locking, the additional lock contention from page faults limits the overall scalability, but the two RCU-based designs scale much better: hybrid locking/RCU and pure RCU outperform read/write locking by 60% and 70%, respectively.

In all three benchmarks, system time on 80 cores drops precipitously with each increasingly concurrent address space design, as shown in Table 1. With read/write locking, system time represents a significant portion of each benchmark's running time. Pure RCU demonstrates between 88% and 94% less system time because of a combination of reduced cache contention, less time manipulating lock wait queues, and less time handling sleeps and wakeups.

7.3 Isolated VM performance

All three application benchmarks spend a substantial fraction of their time performing computation in user space, making it difficult to precisely quantify the performance gains in the virtual memory system. We employ a microbenchmark to isolate the performance of the VM system and closely analyze the costs and scalability of page fault handling. Our microbenchmark uses one core to perform mmap/munmap operations for a given fraction of time (to control how often the mmap_sem is held in write mode), while the remaining cores continuously take soft page faults. This microbenchmark also allows us to drive the VM system much harder than the application benchmarks do, giving a glimpse at how applications may perform at even higher core counts.

Figure 16 shows the base throughput of the microbenchmark without any mmap or munmap operations. In effect, this amplifies the behavior exhibited by Metis and Dedup. Because no locks are acquired in write mode, all non-scalability in this case is the result of cache contention for the cache line holding the mmap_sem. Figure 17 shows the corresponding average cost of a page fault. At 10 cores, page faults take approximately 7,400 cycles in all four designs. By 80 cores, page faults in the three lock-based designs are more than an order of magnitude more expensive. At 80 cores, read/write locking spends 84% of its cycles simply acquiring and releasing the mmap_sem in read mode. The pure RCU design, however, virtually eliminates cache contention; at 80 cores, the page faults take 8,869 cycles, demonstrating near-perfect scalability even at 20 million page faults per second. The few additional cycles are the result of slight non-scalability in the Linux page allocator.

Of course, any real application will also perform memory mapping operations. We simulate this by adjusting how frequently the microbenchmark calls mmap/munmap. For each design, we use enough page faulting cores to drive the design at its peak page

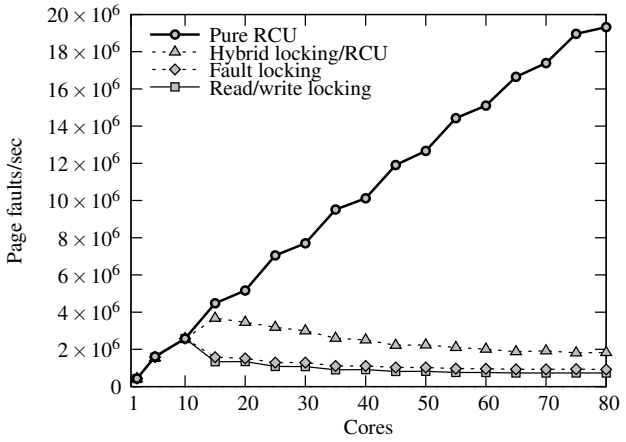


Figure 16. Microbenchmark throughput with no lock contention.

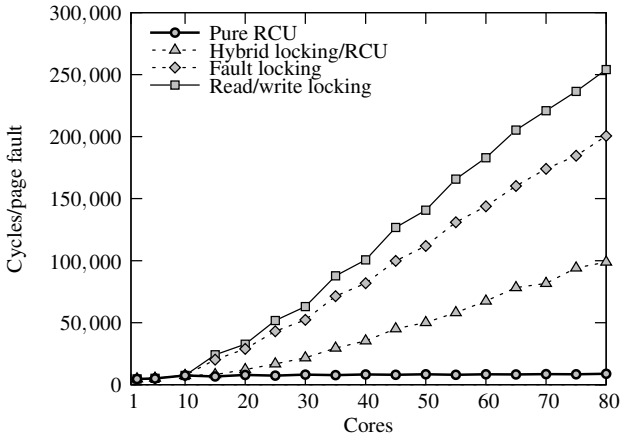


Figure 17. Microbenchmark page fault cost with no lock contention.

fault rate, both to focus on lock contention costs (excluding cache contention costs), and to maximally stress the page fault handler. Figure 18 shows the cost of the page fault handler as we increase the time spent in mmap and munmap. Since each design uses a different number of page fault cores, the base page fault cost—when no time is spent in mmap or munmap—differs slightly, hence we normalize to this base cost. Both read/write locking and fault locking exhibit exponential growth as the probability of lock conflicts increases, ultimately reaching 29× and 21× their base page fault costs, respectively. Hybrid locking/RCU behaves more modestly, as the window for conflicts between memory mapping operations and page faults is restricted to the time the tree lock is held, which is limited to a small fraction of the time that the mmap_sem is held. The pure RCU design again demonstrates near-constant cost, suggesting that the design will continue scaling well on systems that have more than 80 cores.

8. Discussion

We believe the Metis, Psearchy, and Dedup benchmarks highlight the address space concurrency issues faced by the increasingly important class of highly-threaded, VM-intensive applications. The pure RCU design makes Metis and Dedup scale well to 80 cores, but Psearchy suggests the importance of future research to improve write-side parallelism for address spaces. Applications must map memory in order to soft fault it, and, ultimately, the serialization of

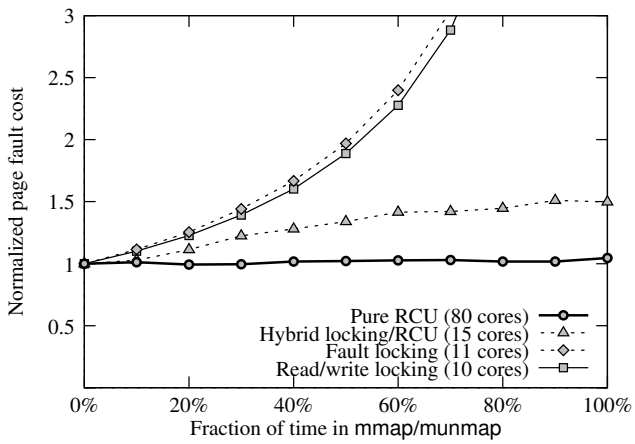


Figure 18. Microbenchmark page fault cost with varying time spent in mmap and munmap, shown relative to the cost with no concurrent memory mapping operations.

memory mapping operations will dominate any parallel component of the virtual memory system.

As the results show, the BONSAI tree allows the pure RCU design to scale at constant cost and, under highly parallel loads, achieve substantially better performance than its locked counterparts. While the BONSAI tree is well suited to storing memory mappings, its design is generic and could benefit other parallel components of the kernel as well as parallel applications. Furthermore, the derivation of the BONSAI tree from a sequential functional data structure and its path-copying optimization are equally applicable to other functional data structures and may yield RCU-compatible versions of other data structures typically protected by locks.

Likewise, little of the pure RCU design is specific to Linux. We believe that the pure RCU approach should apply equally well to other operating systems, and will result in similar scalability benefits.

9. Conclusion

As core counts increase, multithreaded VM-intensive applications can become bottlenecked by kernel operations on their shared address space. To avoid this bottleneck, this paper has introduced an address space design that uses RCU to manage VMAs, page directories, and page tables and uses a new RCU-based balanced tree, BONSAI, to manage the VMA set. Measurements on an 80 core Linux computer show that this design scales better than stock Linux and two other simpler but less-concurrent designs; its use of BONSAI allows the page fault handler to run at a constant cost, independent of the number of cores and of the memory mapping load. As this design applies to any VM system that tracks pages using a balanced tree, it should be applicable to many of the popular shared-memory operating systems.

The source code for our modified Linux kernel is available at `http://pdos.csail.mit.edu/mosbench`.

Acknowledgments

We thank Eddie Kohler, Paul McKenney, and Robert Morris for their feedback. Thanks to Alex Pesterev for getting the 80 core system used in our experiments to work. This work was partially supported by Quanta Computer and by NSF awards 0834415 and 0915164.

References

[1] S. Adams. Implementing sets efficiently in a functional language. Technical Report CSTR 92-10, University of Southampton, 1992.

[2] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[3] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. D. Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proc. of the 8th OSDI*, December 2008.

[4] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proc. of the 9th OSDI*, Vancouver, Canada, October 2010.

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

[6] K. Fraser. Practical lock freedom. Technical Report UCAM-CL-TR-579, Cambridge University, 2003.

[7] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2), May 2007.

[8] M. Herlihy. A methodology for implementing highly concurrent data objects. Technical Report CRL 91/10, Digital Equipment Corporation, October 1991.

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

[10] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6:650–670, December 1981.

[11] Linux Test Project. `http://ltp.sourceforge.net/`.

[12] Y. Mao, R. Morris, and F. Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT CSAIL, May 2010.

[13] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf`.

[14] P. E. McKenney. Sleepable RCU. Available: `http://lwn.net/Articles/202847/` Revised: `http://www.rdrop.com/users/paulmck/RCU/srcu.2007.01.14a.pdf`, October 2006.

[15] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proc. of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[16] P. E. McKenney and J. Walpole. Introducing technology into the Linux kernel: a case study. *SIGOPS Operating Systems Review*, 42(5):4–17, July 2008.

[17] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *Proc. of the Ottawa Linux Symposium*, pages 338–367, July 2001.

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

[19] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proc. of the 4th STOC*, pages 137–142, Denver, CO, 1972.

[20] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222, Dept. of Computer Science, University of Maryland, College Park, 1990.

[21] QEMU. `http://www.qemu.org/`.

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

[23] J. Stribling, J. Li, I. G. Councill, M. F. Kaashoek, and R. Morris. Overcite: A distributed, cooperative CiteSeer. In *Proc. of the 3rd NSDI*, San Jose, CA, May 2006.

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