Read-Copy Update Framework for Userlevel Click

By

Madhuri Venkatesh

A Project Submitted to the Department of Computer Science in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

University of California, Los Angeles, March 2012

Supervised by Professor Eddie Kohler

Abstract

High performance routing lookups are critically important for packet forwarding. Routers must perform a lookup for every arriving packet. As line rates exceed 40Gbps [3], routers are required to perform 125 million lookups a second (assuming a minimum packet size of 40 bytes). Additionally, routers are required to perform sophisticated packet processing providing services which support security, video and mobility [9]. Software routers are more flexible and extensible for such processing when compared to hardware routers. However, software routers are an order of magnitude slower than their hardware counterparts [2]. With multi-core systems becoming more popular and more prevalent, leveraging them is key to improving performance in software routers. However, obtaining a simultaneously correct and fast thread safe solution remains challenging. For example, conventional locking allows only a single thread to access a shared data structure at a time. Although this solution is easily correct, it does not take complete advantage of the available parallelism, and is evidently slower. Reader-writer locks, which allow readers to proceed concurrently with other readers, but not with updaters, are faster in comparison, yet slower than non-locking sequential performance.

We have described an approach for providing thread safety using *Read-Copy Update* (RCU) [10]. RCU is a synchronization technique which scales efficiently on read intensive workloads. It has very low read side overhead, allowing readers to proceed concurrently with other readers and updaters. A typical router workload is read intensive, with 0–5% updates. For such a workload, using a commodity 8 core machine, we have verified that the performance of the RCU approach is near non-locking sequential performance, and is up to 27 times faster than a reader-writer lock.

Acknowledgements

I am tremendously grateful to my advisor, Professor Eddie Kohler for his continued guidance, support and ideas. Working with him was a rare privilege—his patience, attention to detail, knowledge of the subject and eye for both elegance and relevance continue to inspire me. This project was a learning curve for me, and Professor Eddie has supported me through his feedback and insight all along the way.

Rohit Kumar contributed equally to this project. This work reflects his energy and inputs. I've learned from his organized way of working, knack for automation, eye for elegance and brutally honest feedback.

I am thankful to Professor Todd Millstein, Raghu Prabhakar, Aditya Bhiday and Anirudh Udupa for discussions on different aspects of my project.

I am indebted to my parents and sister for their unconditional loving support.

Contents

1 Introduction **4**

2 Background **5**
2.1 An Overview of RadixIPLookup 5
2.1.1 The Radix Tree 5
2.1.2 Route Vector for RadixIPLookup 6

3 The Problem **9**
3.1 Correctness 10
3.2 Updater-Updater Conflicts 10
3.2.1 Case 1: Concurrent updaters use the freelist. 13
3.2.2 Case 2: Concurrent updaters call `push_back()`. 13
3.2.3 Verifying conflicts in Click 15
3.3 Reader-Updater Conflicts 17
3.3.1 Reader-Updater conflicts in the Vector 17
3.3.2 Reader-Updater Conflicts in the Radix Tree 18
3.3.3 Verifying conflicts in Click 18

4 The Solution **21**
4.1 Read-Copy Update 21
4.1.1 Conceptual Overview 21
4.1.2 RCU Using Timers 22
4.1.3 RCU Using The Click Scheduling Loop 23
4.1.4 Protecting against Updater-Updater Conflicts 26
4.1.5 The RCU API for Userlevel Click elements 26
4.2 Reader-Writer Lock 29
4.3 Performance Hypothesis 30

5 Performance Evaluation **30**
5.1 Experimental Setup 30
5.2 Macro-benchmarks 30
5.2.1 Pure Reader Workload 31
5.2.2 Read Intensive Workload With One Writer 32
5.2.3 Write Intensive Workloads 34
5.3 Micro-benchmarks 36

6 Related Work **39**

7 Conclusion **40**

8 Future Work **40**

A Source Code **41**
A.1 Getting the Source Code 41
A.2 Running Benchmarks 41

1 Introduction

The ever-increasing volume of network traffic over the internet demands highly efficient routing lookups. Apart from performing route lookups, routers are also required to support a variety of other functions such as encryption, classification, inspection and filtering [9]. When compared to hardware routers, software routers are more flexible and easily extensible for integrating and adding such functionality. However software routers struggle to comfortably scale beyond 10 Gbps [9], being far outpaced by line rates which exceed 40 Gbps [3].

Route lookups are a significant challenge for software router performance and scalability. A route lookup minimally involves searching a large routing table for the correct destination port per packet. Routing table designs have remained an active area of research [11, 13, 5] as routing tables continue to increase in size [6]. A routing table can have over 366,000 routes [1]. Routing tables are read-write data structures with reads far outnumbering writes. Every forwarded packet involves reading the routing table; updates are far fewer in comparison.

Commodity machines with SMP multi-core architectures are growing in popularity. A software modular router such as Click [16] running on commodity machines can benefit greatly by exploiting parallelism on multi-core architectures. Building a scalable and safe multi-core application is not a trivial task. Shared data structures in a multi-threaded application require synchronization for safe and correct access. Synchronization introduces overheads which attenuate the benefits of parallelism. Locks ultimately have the effect of serialization such that only one thread accesses shared data at a time. As our goal is to support a fast lookup performance, a conventional locking solution which acquires a lock for every route lookup would be inefficient. Reader-writer locks which allow readers to proceed concurrently with other readers have the undesirable property of excluding readers in the presence of an active concurrent updater.

We wish to design a highly efficient multi-core solution for IP lookups in a software router. This workload is typically read intensive—comprising 90% readers and less than 10% updaters. *Read-Copy Update* (RCU) [10] scales efficiently on read intensive workloads. This is because RCU has very low read side overhead, allowing concurrent readers to execute in the presence of a writer without any synchronization. Concurrent writers generally require some form of synchronization. “RCU permits both readers and writers to make forward concurrent progress” [7]. For these reasons, we implemented an RCU framework for the Click modular router. We compare the performance of our approach against a traditional reader-writer lock for a read intensive workload with few updates ($\leq$ 5% of the total workload) and show that RCU performs up to 27 times faster.

The rest of the document is organized as follows. Section 2 gives the reader a background on the userlevel Click elements we are concerned with. Section 3 describes why implementing a multicore solution was challenging. Section 4 describes our solutions to the problem along with the implementation details. Section 5 describes the performance analysis.

2 Background

This section presents a brief overview of the *RadixIPLookup* Element. Our approach involved implementation and analysis using the RadixIPLookup Element. We then extended our solution to other elements in Click.

2.1 An Overview of RadixIPLookup

The *RadixIPLookup* Element [15] is a class which is used to perform IPv4 route lookups. The routing table involves a radix trie to perform route lookups and a vector to store the route information such as the port and gateway. This structure is explained in further detail.

2.1.1 The Radix Tree

The radix tree is structured as a trie. The trie has 2^8 nodes at the first level, and 2^4 nodes in the subsequent 6 levels. A lookup therefore has to traverse a maximum of 7 levels. This structure covers all 2^{32} IP addresses ($2^8 \times (2^4)^6 = 2^{32}$) when the last level is fully occupied. Figure 1 shows the tree pictorially.

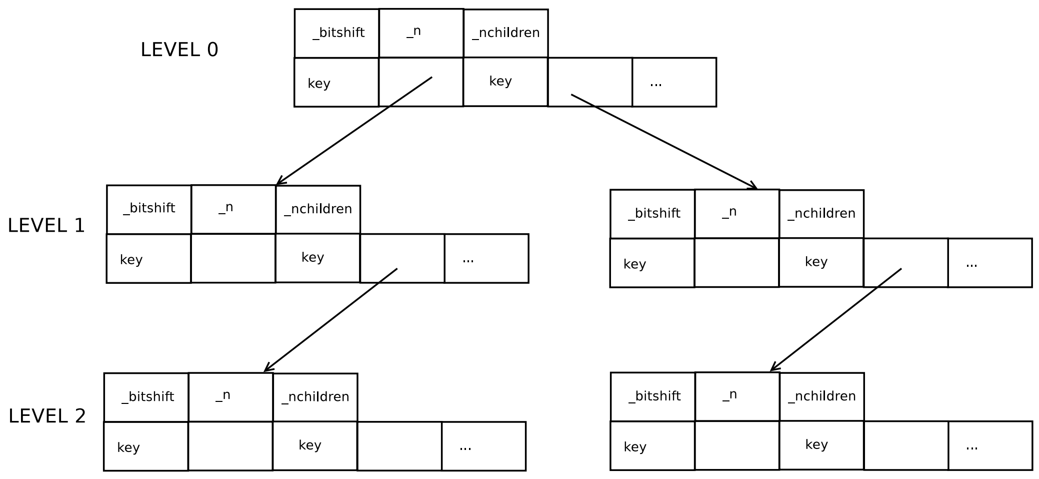


Figure 1: The Radix Tree

Each level consists of radix nodes. The structure of the radix node is shown in Listing 1. `_n` represents the node's fanout (the number of keys it contains). `_bitshift` represents the number of bits to be shifted at that level for prefix matching. For the radix node at level 0, `_n` is 256 and `_bitshift` is 24 (which is $32-8$). This means that the first 8 bits of the address are matched. For the radix nodes at level 1, `_n` is 16 and `_bitshift` is 20 (which is $32-8-4$). This means that at level 1, the first 12 bits of the address have been matched. The functions `add_route()` and `delete_route()` are used to add and delete routes into the routing table. Table 1 breifly summarizes the role of some functions in RadixIPLookup.

```
class Radix {
  private:
    int _bitshift;
    int _n;
    int _nchildren;
    struct Child {
        int key;
        Radix *child;
    } _children[0];
...
...
}
```

Listing 1: The Radix Class

Lookups RadixIPLookup uses a longest prefix match algorithm [19], matching a candidate route with the highest subnet mask. The node at the topmost level stores keys which need to match up to an 8 bit long prefix. The nodes in the next level store keys for routes which need to match up to a 12 bit long prefix, and so on. The last level stores keys which can match up to a 32 bit long prefix. Listing 2 shows the code for the lookup. Given an IP address *addr* which we need to lookup, we calculate the child pointer of the radix node which we need to follow using the *bitshift* at that level. If the key for that child exists, we repeat the process, trying to obtain the longest prefix match. If it does not exist, we have found the longest prefix match for that node, so we return the key value. This key indexes into a vector which will return the gateway and port for the address.

```
  static inline int lookup(const Radix *r, int cur,
                           uint32_t addr) {
    while (r) {
      int i1 = (addr >> r->_bitshift) & (r->_n - 1);
      const Child &c = r->_children[i1];
      if (c.key)
       cur = c.key;
      r = c.child;
    }
    return cur;
  }
```

Listing 2: The lookup function

2.1.2 Route Vector for RadixIPLookup

The Click library files provide an implementation of a generic Vector. A brief explanation of the use of the vector class in RadixIPLookup is described here. The RadixIPLookup Element uses a vector of type IPRoute. IPRoute is a quintuple consisting of *address*, *mask*, *gateway*, *port* and *extra*. *Address* is the IP address of the route. *Mask* is the subnet mask for that route. *Gateway* is the destination to which a packet matching an address/mask will be routed to through the port. *Extra* is used to recycle previously used but currently unused space within the vector. It is used to maintain a list of unused entries in the vector. We call this list the *freelist*. If the index in the vector is currently in use extra is set to -1, otherwise it stores the index of the next entry in the freelist. If an entry in the vector is deleted, the index of that entry is added to a freelist. Table 2 represents a sample vector.

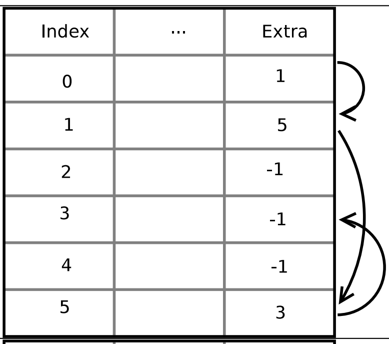
Function name	Description
lookup(const Radix *r, int cur, uint32_t addr)	Performs a longest prefex match lookup for a given IP address. Returns an index into the vector where the route is located.
add_route(const IPRoute &route, bool set, IPRoute *old_route, ErrorHandler*)	Adds the route specified by `route` to the routing table. The IP address which the route corresponds to is included as part of *IPRoute*. The existing route for that IP address is replaced if `set==true`. If an old route exists for that IP address, it is returned in `old_route`.
remove_route(const IPRoute& route, IPRoute* old_route, ErrorHandler*)	Removes a route from the routing table. The address and subnet mask of the route are specified in `route`. If a route matching route is found, it is removed from the table; A pointer to the route is returned in `old_route`.
change()(const Radix *r, int cur, uint32_t addr)	Helper function used by `add_route()` and `remove_route()` to locate and change values in the radix tree.

Table 1: Abridged list of functions in RadixIPLookup.

Index	Addr	Mask	Port	Gw	Extra
0	0.0.0.1	255.255.255.0	1	22.34.198.3	1
1	1.1.0.1	255.255.255.0	1	1.34.198.3	5
2	2.2.2.1	255.255.255.0	1	2.34.198.3	-1
3	1.0.0.1	255.255.255.0	1	9.34.198.3	-1
4	4.0.0.1	255.255.255.0	1	8.34.198.3	-1
5	4.0.0.1	255.255.255.0	1	133.34.198.3	3

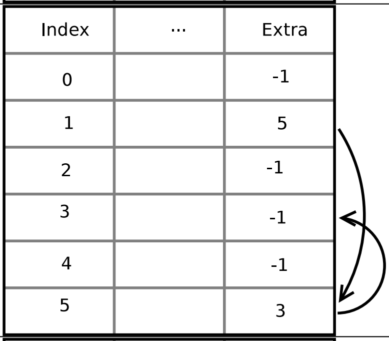
Table 2: Sample vector with routes

State 1: Initial state of the vector. The freelist reads $\{$`_vfree =0`$, 0 \rightarrow 1 \rightarrow 5 \rightarrow 3\}$.



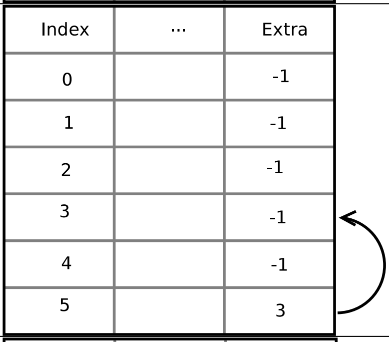
Index	···	Extra
0		1
1		5
2		-1
3		-1
4		-1
5		3

State 2: A route is inserted. The index 0 is reused. The freelist reads $\{$`_vfree =1`$, 1 \rightarrow 5 \rightarrow 3\}$.



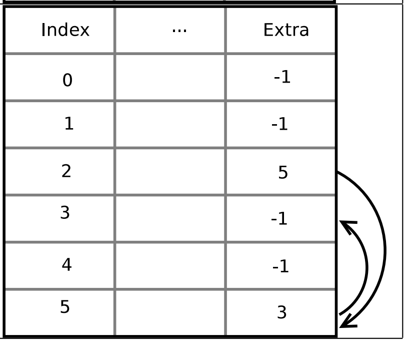
Index	···	Extra
0		-1
1		5
2		-1
3		-1
4		-1
5		3

State 3: Another route is inserted. The freelist now reads $\{$`_vfree =5`$, 5 \rightarrow 3\}$.



Index	···	Extra
0		-1
1		-1
2		-1
3		-1
4		-1
5		3

State 4: Route located at index 2 is removed from the vector. Index 2 is added to the free-list. The freelist now reads $\{$`_vfree =2`$, 2 \rightarrow 5 \rightarrow 3\}$.



Index	···	Extra
0		-1
1		-1
2		5
3		-1
4		-1
5		3

Figure 2: Example illustrating the working of the freelist in the vector. `_vfree` stores the starting index of the freelist. A value of -1 in the extra field indicates that the entry is in use and hence not a part of the freelist. The address, gateway, mask and port fields are not shown.

The use of *extra* is illustrated with the help of an example in Figure 2 for the sample vector shown in Table 2. Reads and deletions are O(1). Insertions are O(1) amortized over the cost of many insertions. This is because the vector is dynamically resized when it is full.

Table 3 describes the members of the vector we discuss in this document. The vector is initialized to hold a specific number of entries (`_capacity`). The number of entries currently in use in the vector is `_n`. The `push_back()` function adds an entry to the vector. This increments `_n` by one. If the vector is full (i.e. `_n` $\geq$ `_capacity`), it is resized to double the value of its previous capacity with a call to `reserve_and_push_back()`.

Member	Description
`_capacity`	The total number of entries (routes) which the vector can store (before the next dynamic resize).
`_n`	The number of entries currently in use.

`push_back``( T* x)`	If there is space in the vector, inserts the value at the end. Otherwise, it resizes the vector and then inserts the value.
	`{`
	`if (_n < _capacity)`
	`   store[_n++]=x;`
	`else`
	`   reserve_and_push_back(`
	`         RESERVE_GROW, &x);`
	`}`

`reserve_and_push_back``( size_type n, const T *x)`	Resizes the vector to hold n more values, and then inserts x.

Table 3: Abridged list of members used in the route vector for RadixIPLookup.

3 The Problem

Our goal is to improve lookup performance by allowing for multi-thread access to Click elements. In this section we look at existing problems which hinder safe multi-thread access.

This section is divided into Updater-Updater Conflicts and Reader-Updater Conflicts. The Updater-Updater Conflicts section describes how multiple updaters can cause race conditions which corrupt the state of shared data structures. The Reader-Updater Conflicts section describes how a reader racing with an updater might read stale or invalid data.

3.1 Correctness

Before analyzing race conditions, we define what we deem acceptable results for concurrent reads and updates.

1. Concurrent updaters exhibit *sequentially consistent* behavior [17].
   The result of concurrent updates is as if the updates were applied in some sequential order.

2. Concurrent reads are *eventually consistent* [20].
   Concurrent readers must receive some valid route. Specifically, if a lookup (read) for an IP address *A* returns a route *R*, then for some duration of time, R was a valid route for IP address *A*. Note that R need not be the most up to date route for IP address *A*—address lookups can result in receiving stale routes for short time periods. However, address lookups (reads) must eventually return the most up to date route.

3.2 Updater-Updater Conflicts

We now deal with race conditions and conflicts arising due to multiple updaters running concurrently using the same instance of RadixIPLookup. When multiple threads use the same instance of RadixIPLookup, they access the same data structures within RadixIPLookup.

Radix Tree The radix tree suffers from Update-Update conflicts. If multiple updaters try to modify the same region of the tree, there is a race in `change()` for the assignment of the child pointer. This could cause some of the updates to be lost and also cause a memory leak. The size of the memory leak can be equal to the depth of the tree times the size of a radix node.

The race described above is illustrated with the help of Listing 3.

```
int
RadixIPLookup::Radix::change(uint32_t addr, uint32_t mask,
                                        int key, bool set)
{
  ...
  if (mask & ((1U << _bitshift) - 1)) {
    if (!_children[i1].child
    && (_children[i1].child = make_radix(_bitshift - 4, 16)))
  ...
  ...
}
```

Listing 3: A snippet of the `change()` function where a race can cause a memory leak. This is illustrated in Figure 3.

The `change()` function in Listing 3 is used to create a radix node for a route with a specific address and mask. The key passed as a parameter is the index into the vector where the route is stored. Two or more concurrently executing threads create race conditions which can lead to memory leaks. This is explained pictorially in Figure 3. Looking at the assignment, we see that if there are multiple threads executing `change()`, there is a race for the assignment of the radix node. Multiple threads may

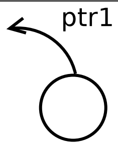
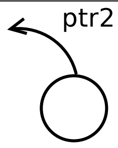
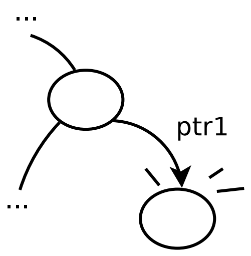
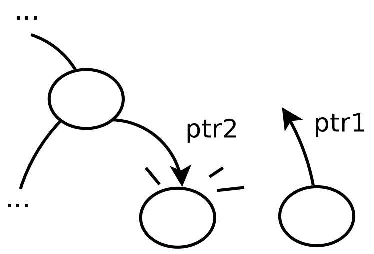
Thread 1	**Thread 2**
`!_children[i1].child` evaluates to true.	`!_children[i1].child` evaluates to true.
Calls `make_radix()`.	Calls `make_radix()`.
 `make_radix()` returns pointer ptr1 to a newly created radix node.	 `make_radix()` returns pointer ptr2 to a newly created radix node.
 `_children[i1].child = make_radix(_bitshift - 4, 16).` The node is added to the radix tree.	 `_children[i1].child = make_radix(_bitshift - 4, 16).` The node is added to the tree. We have a memory leak since the older node is now orphaned. Additionally, Thread 1's update is lost.

Figure 3: Example illustrating the race in the `change()` function. The `change()` function is used in the insertion of new radix nodes. Two concurrent threads insert a radix node into the tree leading to a memory leak.

call `make_radix()`, but only one pointer which is returned by `make_radix()` is assigned.

Vector When there are multiple updaters there is contention for acquiring an index into the vector. Many updaters trying to add a route might receive the same index value. If multiple updaters are given the same value of the index, there can be an inconsistency in the state of the vector. For example, the size of the vector might be stored wrongly or the freelist might be updated wrongly. Since the vector is resized dynamically, we could lose updates and have memory leaks.

We explain the conflicts mentioned above with examples using the `add_route()` function shown in Listing 4. We consider two cases, one in which the concurrently ex-

```
int
RadixIPLookup::add_route(const IPRoute &route,
                         bool set,
                         IPRoute *old_route,
                         ErrorHandler *)
{
  int found = (_vfree < 0 ? _v.size() : _vfree), last_key;
  if (route.mask) {
    uint32_t addr = ntohl(route.addr.addr());
    uint32_t mask = ntohl(route.mask.addr());
    last_key = _radix->change(addr, mask, found + 1, set);
  } else {
    last_key = _default_key;
    if (!last_key || set)
      _default_key = found + 1;
  }
  if (last_key && old_route)
    *old_route = _v[last_key - 1];
  if (last_key && !set)
    return -EEXIST;
  if (found == _v.size())
    _v.push_back(route);
  else {
    _vfree = _v[found].extra;
    _v[found] = route;
  }
  _v[found].extra = -1;
  if (last_key) {
    _v[last_key - 1].extra = _vfree;
    _vfree = last_key - 1;
  }
  return 0;
}
```

Listing 4: The add_route function

ecuting threads retrieve an index from the freelist, the other in which the concurrently executing threads make a call to `push_back()` on the vector.

3.2.1 Case 1: Concurrent updaters use the freelist.

We now illustrate how a race condition can corrupt the state of the vector. The same example is illustrated pictorially in Figure 4. Assume the freelist is as follows before the execution of this code sequence:
Initially the freelist reads $\{$`_vfree =3,` $3 \rightarrow 1 \rightarrow 5\}$. Consider the following sequence of events:

1. Thread 1 calls `add_route(Y,1,NULL,NULL)` with `Y.addr = A`, `Y.mask = M`.
2. Thread 2 calls `add_route(Y,1,NULL,NULL)` with `Y.addr = A`, `Y.mask = M`.
3. Updater 1 and Updater 2 execute line 4 concurrently and recieve the same value of *found*. Let us assume that *found* is some key within the vector and is not equal to *_v.size()*. Updater 1 executes line 7 first. This updates the key in the radix tree to be the value *found* which is 3 according to our example.
4. Updater 2 executes line 7. The value of the key is set to *found* which is 3 since the freelist has not changed. (See Figure 4).
5. Updater 1 executes the remaining code in `add_route()` including line 27 which sets `_v[found].extra` to -1 (which means that the index is in use). After the Updater 1 has finished executing add_route(), the freelist reads $\{$`_vfree =1,` $1 \rightarrow 5\}$.
6. Updater 2 executes the remaining code sequence. It executes line 24 which sets `_vfree` to `_v[found].extra`. `_v[found].extra` has been set to -1 by Updater 1. Thus `_vfree` is now -1.
   The state of the freelist is now $\{$`_vfree = −1,` $1 \rightarrow 5\}$. The correct state is $\{$`_vfree = 1,` $1 \rightarrow 5\}$. If `_vfree` is -1, it means that there are no more free keys in the vector, which is not true. This sequence brings the vector into an inconsistent state. As a result, the routing table is also in an inconsistent state.

3.2.2 Case 2: Concurrent updaters call `push_back()`.

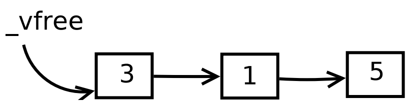
Now consider the case when `_v.push_back()` is executed concurrently. For this we will look at the Click code from vector.cc in Listing 5.

```
template <class T> inline void
Vector<T>::push_back(const T& x)
{
  if (_n < _capacity) {
    new(velt(_n)) T(x);
    ++_n;
  } else
    reserve_and_push_back(RESERVE_GROW, &x);
}
```

Listing 5: The push_back() function

We first consider the case when there is enough space in the vector. Updater 1 and Updater 2 use the same value of `_n`. The updater which wins the race installs its key

Initial state of the freelist:



Thread 1	**Thread 2**
calls `add_route(X,1,NULL,NULL)`	calls `add_route(Y,1,NULL,NULL)`

Executes:
 	
`found= (_vfree < 0 ?`
`        _v.size():_vfree)`
 	
Obtains a value `found= _vfree= 3`
	Executes:
	
	`found= (_vfree < 0 ?`
	`        _v.size():_vfree)`
	
	Obtains a value `found= _vfree= 3`

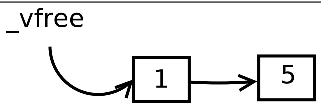
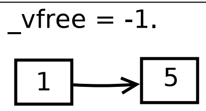
Executes: ... `_v[found].extra = -1;` As a result, the freelist is updated as shown in the figure. 	Thread is blocked or paused or context-switched.
Thread exits `add_route()`.	Executes: `_vfree = _v[found].extra`. `_v[found].extra` was set to -1 by thread 1. So `_vfree` is -1. Freelist is now updated as shown in the figure. This is wrong since `_vfree` is -1 although there are unused indices in the freelist. A value of -1 for `_vfree` indicates that the freelist is empty which is untrue. 

Figure 4: The example illustrates how two concurrent updaters can corrupt the sate of the freelist if they call `add_route()`. The initial state of the freelist is as shown in the figure. The first parameter is the route, the second parameter asks the function to use the route supplied even if a route previously existed for that IP address. The third parameter *old_route* retrieves a preexisting route for that IP if it is not set to NULL. The fourth parameter is the error handler.

at position i, and the other update is lost. We now consider the case when there is not enough space in the vector. The expression `_n < _capacity` evaluates to false. Some of the race conditions here could bring the vector to an inconsistent state or cause memory leaks. They are better understood with the help of the graphic example in Figure 5 and the `reserve_and_push_back()` function shown in Listing 6. The assignment of `new_l` in line 15 of Listing 6 is a race condition which can lead to a memory leak. The size of the leak is equal to the size of `new_l`. The other consequence is that one update will be lost. If both the updaters call `CLICK_LFREE()` in line 14, we could have a segmentation fault.

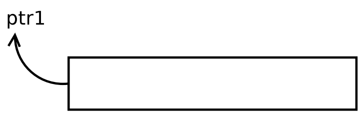
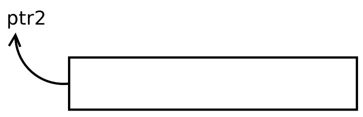
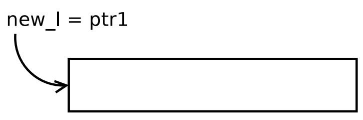
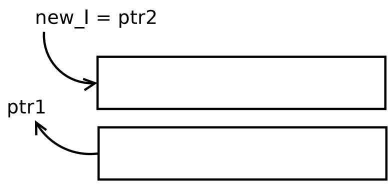
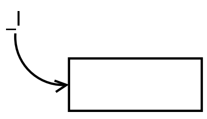
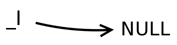
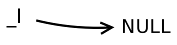
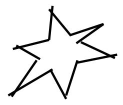
Thread 1	**Thread 2**
Calls `reserve_and_pushback()`.	Calls `reserve_and_pushback()`.
 `CLICK_LALLOC()` returns pointer ptr1 to a newly allocated vector.	 `CLICK_LALLOC()` returns pointer ptr2 to a newly allocated vector.
 `new_l = ptr1.` The node is added to the radix tree.	 `new_l = ptr2.` We have a memory leak since the older node is now orphaned.
 `CLICK_LFREE(_l, ...).` The pointer to the old vector is freed. 	 `CLICK_LFREE(_l, ...).`  Potential failure or memory corruption since the thread attempts to free a pointer which is already freed.

Figure 5: Example illustrating the race during a dynamic resize of the vector.

3.2.3 Verifying conflicts in Click

Click scripts were used to verify existing race conditions for checking correctness of our solutions. New elements were created and run in multi-threaded mode. The

```
Vector<T>::reserve_and_push_back(size_type want,
                                 const T *push_x)
{
  if (want < 0)
    want = (_capacity > 0 ? _capacity * 2 : 4);
  if (want > _capacity) {
    T *new_l = (T *) CLICK_LALLOC(sizeof(T) * want);
    if (!new_l)
      return false;
    for (size_type i = 0; i < _n; i++) {
      new(velt(new_l, i)) T(_l[i]);
      _l[i].~T();
    }
    CLICK_LFREE(_l, sizeof(T) * _capacity);
    _l = new_l;
    _capacity = want;
  }
  if (unlikely(push_x))
    push_back(*push_x);
  return true;
}
```

Listing 6: reserve_and_push_back()

StaticThreadSched() Element was used to specify the thread on which the Element must be run. Listing 7 facilitates a clearer picture of the use of *StaticThreadSched()* for this case. We created Click elements called BashRadixIPLookup and PoundRadixIPLookup which repeatedly add and delete the same route with the same address but different values of the port. This is illustrated in Table 4. The parameters for `add_route()` mentioned in the listing are simplified. Both BashRadixIPLookup and PoundRadixIPLookup run concurrently on the same instance of RadixIPLookup. This concurrency is ensured by using the *StaticThreadSched()* Element in the script.

```
Idle
 -> r :: RadixIPLookup
 -> Idle;

writer0 :: PoundRadixIPLookup(r);
writer1 :: BashRadixIPLookup(r);

StaticThreadSched(
        writer0 0,
        writer1 1,
);

DriverManager(wait 3, print r.table, stop);
```

Listing 7: Click script for verifying updater-updater conflicts.

The tests resulted in segmentation faults and assertion failures, which did not occur if locks were acquired during updates.

```
bool
BashRadixIPLookup::
run_task(Task *) {
  ...
  for( i=0; i<10000, i++) {
    add_route (A,X);
    delete_route (A,X);
  }
 ...
}
```

```
bool
PoundRadixIPLookup::
run_task(Task *){
  ...
  for(i=0;i<10000,i++){
    add_route (B, Y);
    delete_route (B,Y);
  }
  ...
}
```

Table 4: The `run_task()` methods of BashRadixIPLookup and PoundRadixIP-Lookup.

3.3 Reader-Updater Conflicts

The conflicts in the RadixIPLookup Element occur primarily in two data structures: the vector used to store the routes, and the radix tree which stores a key indexing into the vector. This section describes the conflicts which occur in each of these data structures.

3.3.1 Reader-Updater conflicts in the Vector

We look at reader-updater conflicts which involve read delete races, and discuss their consequences. The race conditions are described in the context of the `add_route()` function which adds a route to the radix tree. The way an index is obtained for insertion is summarized using the pseudo-code in Listing 8.

```
if(free_list is not empty) {
  index = free_list; free_list = free_list->next;
  _v[free_list.index] = val;
} else
  _v.push_back(val);
```

Listing 8: Pseudo-code for acquiring an index in `add_route()`

We can observe at least two inadmissible conflicts which can be caused by race conditions. Firstly, if two updaters and a reader run concurrently, the reader might get an index into an entry which has been freed and reused. The reader may end up receiving a route which is incorrect for the IP address in question. Figure 7 illustrates this race. BashRadixIPLookup and PoundRadixIPLookup are two updaters which update the routing table by adding a route. ReadRadixIPLookup performs a lookup for IP address A. There is a chance that the lookup returns a route which was never a valid route for IP address A. This does not observe the correctness policy described in Section 3.1. Secondly, a reader might access invalid memory in the presence of a concurrent updater. One must recall that the `push_back()` operation dynamically resizes the vector if necessary. If an updater initiates a dynamic resize operation amidst a concurrent reader, the reader might access invalid memory.

3.3.2 Reader-Updater Conflicts in the Radix Tree

This section describes some of the reader-updater conflicts which exist within the radix tree and justifies why they are not a source for concern.

We will now look at a race condition which can occur in the radix tree when a reader and an updater execute concurrently. Consider a case where we have a reader and an updater running concurrently. The updater deletes route X which holds information for IP address A, the reader performs a lookup on the very route A.
We have two threads one running: an updater thread trying to remove a route X from

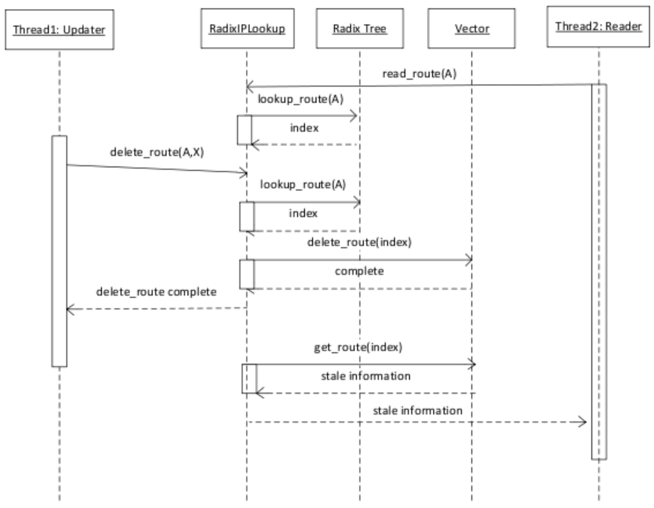


Figure 6: Race condition with a concurrent reader and updater

the routing table, and a reader thread performing a lookup for IP address A. The reader might receive stale information for A. This happens if the reader acquires a reference to the radix node containing Route X before the pointer is removed from the tree. The race is illustrated with the help of the sequence diagram in Figure 6. However, readers which execute after the updater has finished will not receive stale data for Route X. These results are acceptable according to our correctness requirement specified in 3.1. This is justified by the argument that the reader received a stale route: it was once valid for the given address, and the ultimate price paid is the price of one wrong lookup, which as we said above, is acceptable by our policy.

3.3.3 Verifying conflicts in Click

Some of the conflicts mentioned in Section 3.3.1 were verified by running Click scripts. Our approach for verifying reader-updater conflicts is the similar to the one mentioned in Section 3.2.3. Click elements BashRadixIPLookup, PoundRadixIPLookup repeatedly update the routing table. ReadRadixIPLookup repeatedly performs a route lookup.

```
int
RadixIPLookup::remove_route(const IPRoute& route,
                            IPRoute* old_route,
                            ErrorHandler*)
{
  int last_key;
  if (route.mask) {
    uint32_t addr = ntohl(route.addr.addr());
    uint32_t mask = ntohl(route.mask.addr());
    // NB: this will never actually make changes
    last_key = _radix->change(addr, mask, 0, false);
  } else
    last_key = _default_key;
  if (last_key && old_route)
    *old_route = _v[last_key - 1];
  if (!last_key || !route.match(_v[last_key - 1]))
    return -ENOENT;
  _v[last_key - 1].extra = _vfree;
  _vfree = last_key - 1;
  if (route.mask) {
    uint32_t addr = ntohl(route.addr.addr());
    uint32_t mask = ntohl(route.mask.addr());
    (void) _radix->change(addr, mask, 0, true);
  } else
    _default_key = 0;
  return 0;
}
```

Listing 9: The remove_route() function

Table 5 shows how these elements are used to verify conflicts. ReadRadixIPLookup, PoundRadixIPLookup and BashRadixIPLookup run concurrently on an instance of RadixIPLookup. A and B are the addresses, X and Y are the values of the port. The least-common ancestor of A and B is the root node. Locks are acquired during updates since we want to verify reader-updater conflicts as opposed to updater-updater conflicts. All three of these are run concurrently on the same instance of RadixIPLookup. The race is illustrated with the help of a sequence diagram in Figure 7.

A thread-safe implementation will have us expect that we always get X or the default route. A default route is returned if there was no prefix matching the route being looked up in the tree. But we see that the lookup sometimes returns Y, which is incorrect.

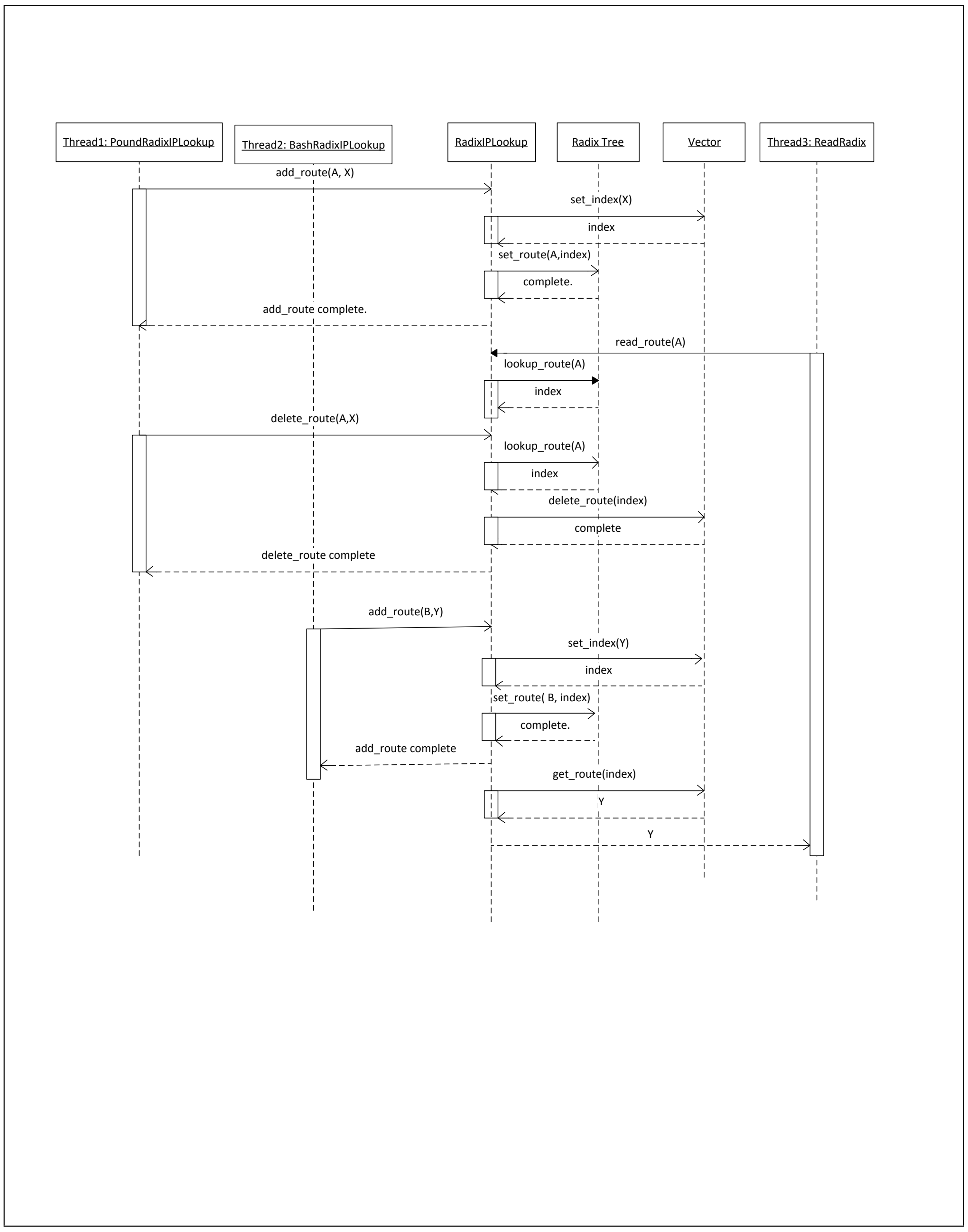


Figure 7: Race condition with a concurrent reader and updater

```
bool
BashRadixIPLookup::
run_task(Task *) {
  ...
  for( i=0; i<10000, i++) {
    add_route (A,X);
    delete_route (A,X);
  }
 ...
}
```

```
bool
PoundRadixIPLookup::
run_task(Task *){
  ...
  for(i=0;i<10000,i++){
    add_route (B, Y);
    delete_route (B,Y);
  }
  ...
}
```

```
bool
ReadRadixIPLookup::
run_task(Task *){
  ...
  for(i=0;i<10000,i++){
    lookup_route (A);
  }
  ...
}
```

Table 5: The `run_task()` methods of BashRadixIPLookup, PoundRadixIPLookup and ReadRadixIPLookup.

4 The Solution

The race conditions described in the previous sections occur due to threads accessing shared data structures. RCU is known to be an efficient synchronization technique for read-heavy workloads. We first describe our solution using RCU. We then describe our solution using a reader-writer lock.

4.1 Read-Copy Update

4.1.1 Conceptual Overview

RCU allows readers to proceed concurrently with an updater. The updater can **remove** shared data structures, but is not allowed to **delete** it. Multiple copies of shared data structures are maintained. The old copies are deleted when they are no longer being used. A thread reaches a quiescent state when it releases references to shared data structures. A grace period is reached when all threads have undergone at least one quiescent state. It is safe to delete stale copies (reclaim memory) during a grace period. Figure 8 illustrates this concept.

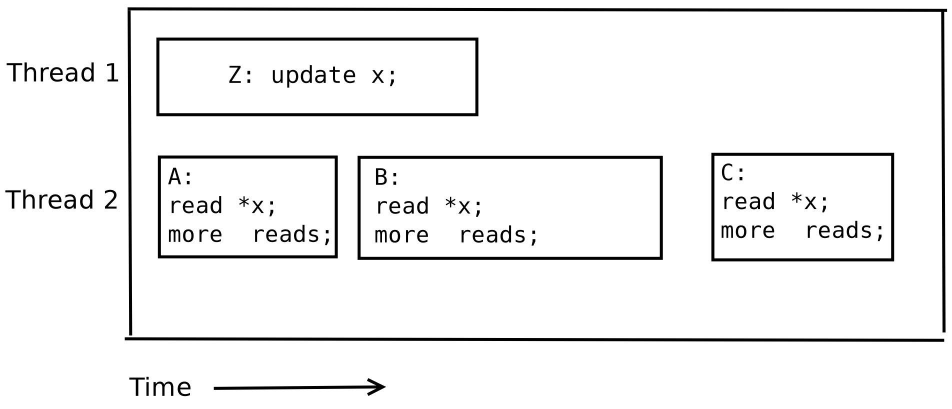


Figure 8: Thread 1 and Thread 2 begin concurrent execution. A, B and C represent read-side critical sections which read the pointer *x*. Z is an update-side critical section which updates the pointer *x* by deleting the old pointer. A and B access *x* when Z is in progress. We allow Z to remove the old pointer but we do not allow Z to **free** it. C reads the value of *x* **after** Task Z has finished; it is not dependent on the older value of *x*. It is safe to delete the older copy of *x* after A and B have completed.

4.1.2 RCU Using Timers

IP lookups are read-side critical sections. An IP lookup reads the routing table which is a shared data structure. IP lookups using RadixIPLookup finish within 1000 ns on modern commodity machines [14]. A fixed time interval $\geq$ 1000 ns can be a grace period to free stale data-structures. An interval of 1000 ns leads to very frequent reclamations. A very wide time interval can lead to infrequent reclamations, and consequently, a larger memory footprint. So a convenient interval of 500 ms is chosen.

Figure 9 explains the working of the timer mechanism. Our implementation involved using the Click Timer class to schedule reclamations. Click provides a Timer class which calls functions at specified time intervals. We use this to schedule a callback which reclaims freed data at a time interval of 500 ms.

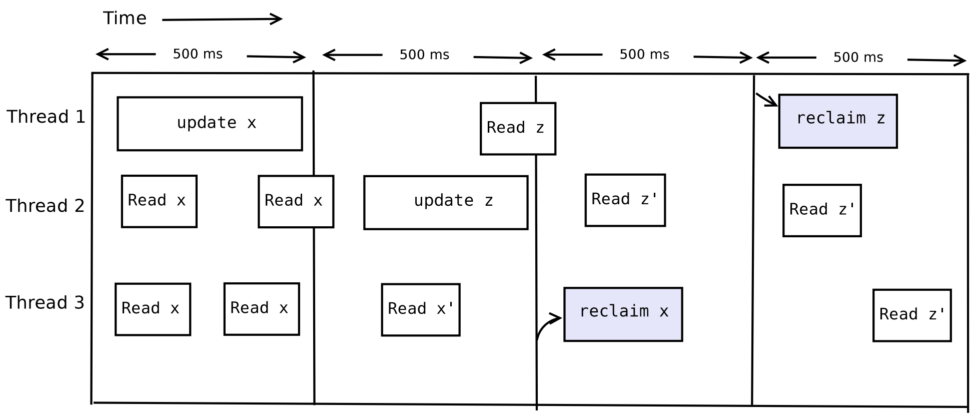


Figure 9: Three threads run concurrently. In the first 500 ms interval, an updater changes a shared data structure. Readers which have started before the update completes recieve the older value of *x*. Readers which begin after the updater completes recieve the newer value. In each interval, stale data from the intervals prior to the previous interval can be reclaimed. It is safe to reclaim *x* during or after the third interval and safe to reclaim *z* during or after the fourth interval. Reclamations are highlighted in gray.

4.1.3 RCU Using The Click Scheduling Loop

The timer expiry as a quiescent state described in Section 4.1.2 works well for RadixIPLookup since each lookup completes within a fixed time interval. For other elements in Click, it may not be possible to determine the duration of a read-side critical section. We wish to detect quiescent states for all other userlevel elements without having to know how long it takes for a reader task to complete. The scheme described here does not require that the reader task be of a specific duration. It does require that the reader task eventually terminate. The approach involves using the scheduling loop in Click to detect quiescent states.

The Click RouterThread Driver Loop A *task* in Click is something that requires CPU time or packet processing time. A task can involve performing an IP lookup, encapsulating a packet, etc. Click has a class called *RouterThread* which is responsible for running tasks. When Click is allowed to run with say *N* threads, *N* instances of *RouterThread* are created. Each *RouterThread* instance runs a `driver()` function which runs tasks scheduled by Click elements in a loop. We will refer to this loop as the "driver loop". After running a specified number of tasks, control returns to the driver loop. A highly abridged version of the driver loop is shown in Algorithm 1.

Algorithm 1 The RouterThread Driver Loop

```
$forall$ threads $t_i$ :
while $\exists$ a task for thread $t_i$ do
    Run a task.
end while
```

Grace Periods and Reclamation We describe our approach with the help of Figure 10 and in more formal terms using Algorithm 2. Each *RouterThread* instance maintains a local epoch counter, *le*. The local epoch counter can either be zero or one. The application has a global epoch counter, *GE*. The global epoch counter is not physically stored; we only use it to explain and justify our approach. Initially all local epoch counters are set to one. After a thread completes a task, it resets its local epoch number. When the local epoch counters of all threads are reset, we reach a grace period. (All threads have released references to shared data structures they once held.) At this point, we reclaim stale data structures from the penultimate global epoch, increment the global epoch counter, set all local epoch counters and continue.

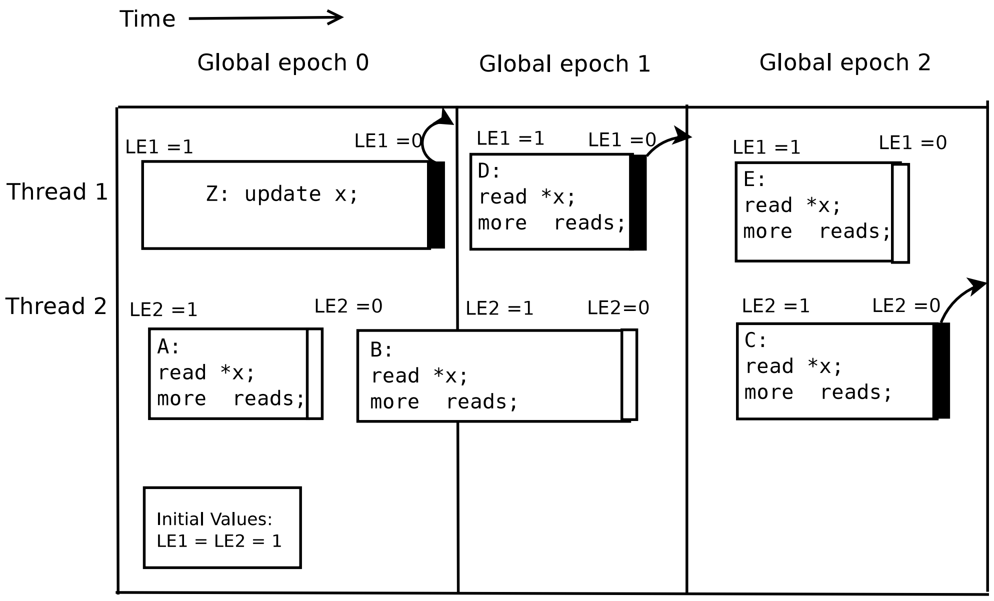


Figure 10: Thread 1 and Thread 2 execute concurrently. The white rectangle indicates that a thread has updated its epoch counter and attempted to reclaim, but was unsuccessful. The black rectangle indicates that a thread has updated its epoch counter and has reclaimed successfully. Following a reclamation, all local epoch counters are reset to 1. It is only safe to free *x* **after** *global epoch 1* has passed. Task Z of *Thread1* modifies the pointer *x* in *global epoch 0*. Task A and B dereference *x* during *global epochs 0* and *1*. For this reason it is not safe to free *x* in *global epoch 0* or *1*. Task C reads *x* in *global epoch 2*, **after** the updater has finished. For this reason, it receives the value of *x* updated by task Z. We advance the *global epoch counter* when we reach a grace period, i.e. when all threads cease to hold references to shared data structures.

The transition between *global epoch 1* and *global epoch 2* in Figure 10 is a grace period for updaters which finish in *global epoch 0*. This is because any readers which start in *global epoch 0* have finished by the end of *global epoch 1*. It is safe to reclaim memory in *global epoch 2*. We can extend this observation using Figure 2 and say that stale data accumulated in *global epoch N* can be reclaimed during *global epoch N+2*. Note that we cannot do this during *global epoch N+1* since a reader which starts in

Algorithm 2 Algorithm for detecting grace periods.

```
Each thread $t_i$ has a local epoch number $le_i$.
The application has a global epoch number $GE$.
Initially $GE = 0$ and $\forall t_i, le_i = 1$.
$forall$ threads $t_i$ :
while $\exists$ a task for thread $t_i$ do
   Run a task.
   set local epoch $le_i = 0$.
   if $\forall t_i, le_i = 0$ then
      Set $GE = GE + 1$.
      Reclaim stale data from $GE - 2$.
      $\forall t_i$,Set $le_i = 1$.
   end if
end while
```

global epoch N might not finish until the end of *global epoch N+1*.

In order to facilitate reclamation of stale data accumulated during *global epoch N* in *global epoch N+2*, we maintain two lists: a *reclaim_now* list and a *reclaim_later* list. When an updater modifies a shared data structure, we maintain the updated copy as well as the older copy. The older copy is also added to the *reclaim_later* list. Whenever a grace period is reached:

1. Reclaim everything in the *reclaim_now* list. The *reclaim_now* list becomes empty after this step completes.
2. Swap the *reclaim_now* list and the *reclaim_later* list.
   The *reclaim_later* list becomes empty and the *reclaim_now* list has the contents of the previous *reclaim_later* list after this step completes.

RCU guarantees We explain how the two RCU guarantees expressed in [7] can be achieved with the help of our scheme. The *Grace Period Guarantee* says that a given read-side critical section cannot extend beyond both sides of a grace period. The *Publication Guarantee* says that readers will see consistent newer versions of the data structures updated by writers. These guarantees are important since they show that readers always access consistent copies of data structures.

1. *Grace-Period Guarantee:* This guarantee states that a given read-side critical section cannot extend beyond both sides of a grace period. We now show that we can maintain this as an invariant in our scheme.

   Any outstanding tasks in global epoch $GE = N - 2$ have finished before the start of global epoch $GE = N$.
   Consider any task which started in $GE = N - 2$,
   At the point when the task starts, the local epoch le_i can either be 0 or 1.

   If $le_i == 1$:
      GE can move to $N - 1$ only after all le_i have been reset to 0.
      le_i is reset to zero only after the task finishes.
      $\therefore$ the task finishes before $GE = N - 1$, it finishes before $GE = N$.

If $le == 0$:
   le_i is set when the global epoch progresses to $GE = N - 1$.
   le_i is reset before GE progresses from N-1 to N.
   le_i can reset only after this task finishes.
   $\therefore$ the task finishes before $GE = N$.

2. *Publication Guarantee:* Although readers proceed concurrently with writers, readers must see either the old version of the data or the newer version. Readers should not see an inconsistent state of the data structure. Readers which begin after an updater has completed should see the newer version. The update side critical section should ensure that readers see a consistent state of mutable data structures. For example, in order to ensure that readers do not see an inconsistent version in RadixIPLookup, variables which reflect the state of the vector such as `_capacity` and `_n` are updated only after the pointers are updated. Memory barriers are also used to ensure that these variables reflect their newer values only after memory has been allocated.

We observe that our approach is similar to the *fuzzy barrier* approach described by Hart et al. [12]. In our scheme, the barrier is the condition which checks that all local epoch variables are set to zero. The barrier protects access to the reclamation code. This barrier is fuzzy because the threads resume execution of other tasks if they are unable cross the barrier and reclaim (non-blocking).

4.1.4 Protecting against Updater-Updater Conflicts

RCU allows concurrent readers to proceed amidst other readers or a concurrent updater. Concurrent updaters require some form of synchronization. We use locks around update-side critical sections to synchronize concurrent updaters.

4.1.5 The RCU API for Userlevel Click elements

We describe the API for userlevel elements, which can be used by an Element to provide RCU synchronization. This section describes how an Element can use the API to provide RCU synchronization, as well as how the API interfaces with the driver loop in Click.

Requirements for Userlevel elements The ReclaimHook class (Listing 10) has been added to facilitate RCU synchronization for userlevel elements. The framework identifies *when* it is safe to reclaim. It is the responsibility of the userlevel Element to identify mutable data structures, and provide callbacks which correctly reclaim memory. A userlevel Element which seeks to use RCU synchronization using our framework is required to do the following:

1. The Element should declare a ReclaimHook variable as a member of the class (for example, declare `ReclaimHook _reclaimhook`).

2. The Element should provide a `reclaim()` method as a member function or as a member of a subclass. The instance of the class containing the `reclaim()` function is the *Reclaimable* object.
   This function should reclaim stale data from the `reclaim_later` list when called, using the mechanism described in Section 4.1.3.

3. The ReclaimHook should be initialized using `initialize(this)` before any reclamations are scheduled.
4. Reclamations can be scheduled or unscheduled using the `schedule()` or `unschedule()` methods respectively.
5. Update-side critical sections must be protected using locks (unless a single updater thread is used).

```
class ReclaimHook:public Hook {

  public:
    inline ReclaimHook(Reclaimable *);
    inline ~ReclaimHook() {}

    inline void initialize(Element *);
    inline bool scheduled()  { return _is_scheduled; }
    inline void schedule()   { _is_scheduled = true; }
    inline void unschedule() { _is_scheduled = false; }
    inline void fire();

  private:
    Element *_owner;
    void * _thunk;
    bool _is_scheduled;
};

inline
ReclaimHook::ReclaimHook(Reclaimable *r)
    :_thunk(r), _is_scheduled(false){

}

void
ReclaimHook::initialize(Element *owner) {
    assert(owner);
    Router *router = owner->router();
    router->master()->add_reclaim_hook(this);
    _owner = owner;
}

void
ReclaimHook::fire() {
  if(_thunk) {
    ((Reclaimable *)_thunk)->reclaim();
  }
}
```

Listing 10: The ReclaimHook class which allows elements to schedule reclamations.

API Usage in the Driver Loop An element using our API registers a callback function (`reclaim()`), which reclaims stale data belonging to that Element, as described previously. When a grace period is reached, it is safe to reclaim stale data accumulated two epochs prior to the current epoch, as discussed in Section 4.1.3. When we reach a grace period, we invoke all registered callbacks using the `ReclaimHook::fire()` method. This is illustrated in Listing 11. We can see that after a grace period, all elements which had scheduled reclamations have had a chance to reclaim stale data.

```
void
Master::try_reclaim()
{
    // multiple threads call try_reclaim at the same time,
    // but only one should excecute it.
    bool got_lock = _try_reclaim_lock.attempt();
    if(got_lock == false)
        return;

    bool reclaim = true;
    RouterThread *t;
    // The condition for a grace period is that for all threads:
    // either the thread has witnessed a quiescent state
    // ( i.e.  _thread_epoch_counts[thread_id] == 0 )
    // or the thread is blocked.
    int n = nthreads();
    for(int i = 0; i < n ; i++) {
        t = thread(i);
        int state = t->thread_state();
        if(_thread_epoch_counts[i] ==1 &&
           (state == RouterThread::S_RUNTIMER ||
            state == RouterThread::S_RUNSELECT ||
            state == RouterThread::S_RUNSIGNAL ||
            state == RouterThread::S_RUNSELECT ||
            state == RouterThread::S_RUNTASK)) {
            reclaim = false;
            break;
        }
    }

    if(reclaim){
        // reclaim memory
        for(int i = 0; i < _reclaim_hooks.size(); i++) {
            if(_reclaim_hooks[i]->scheduled())
                _reclaim_hooks[i]->fire();
        }
        // reset the _thread_epoch_count variables for each thread.
        asm volatile ("" : : : "memory");
        for(int i = 0; i < n; i++) {
            t = thread(i);
            _thread_epoch_counts[i]= 1;
        }
    }
    _try_reclaim_lock.release();
}
```

Listing 11: Invoking callbacks for reclamations using the API.

4.2 Reader-Writer Lock

We have compared the performance of the RCU approach with that of a reader-writer lock. We chose a reader-writer lock since the typical workload for a router is read intensive with few updates. In the absence of an updater, all readers can access shared data without contending for the lock.

Our approach involves locking around the entire read-side or update-side critical section. This is illustrated in the `add_route()` function shown in Listing 13, where a lock is acquired around the entire update-side critical section. Similar locking is used around the read-side critical section in `lookup_route()` (Listing 12).

```
int RadixIPLookup107::lookup_route(IPAddress addr,
                                   IPAddress &gw) const
{
    _lock.acquire_read();
    ...
    read side critical section
    ...
    _lock.release_read();
    return port;
}
```

Listing 12: Reader-writer lock usage in lookup_route()

```
int
RadixIPLookup107::add_route(const IPRoute &route,
                            bool set,
                            IPRoute *old_route,
                            ErrorHandler *)
{
  _lock.acquire_write();
  ...
  write-side crtical section
  ...
  _lock.release_write();
  return 0;
}
```

Listing 13: Reader-writer lock usage in add_route()

Feature	Value
CPU	Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
Number of Cores	8
Operating System	Mac OS X 10.7.2 11C74
Cache-line size	64 bytes
L1 cache size (per-CPU)	32 KB
L2 cache size (shared by 2 CPUs)	256 KB
L3 cache size (shared by 8 CPUs)	8 MB
Main memory size	16 GB

Table 6: Machine configuration.

4.3 Performance Hypothesis

RCU is known to work best for a reader-heavy workload with a small number of updaters. Our claim is that the RCU mechanism for click outlined above has almost zero reader-side overhead. Readers being lock-free and wait-free, we expect RCU performance to be comparable to the original version when there are only readers.

Updaters acquire a lock, so there is some performance penalty in the presence of updaters. We expect the RCU performance for an update-intensive workload to be comparable to that of a Reader-Writer lock.

5 Performance Evaluation

5.1 Experimental Setup

We analyzed the performance of our solutions on the machine whose configuration is shown in Table 6. We used the CPU time reported by system utility */usr/bin/time* to measure performance.

Macrobenchmarks in Section 5.2 evaluate the performance of RCU over a routing table consisting of roughly 167,000 routes. Microbenchmarks in Section 5.3 evaluate the performance of RCU with a over a small range of IP addresses.

5.2 Macro-benchmarks

The macrobenchmarks are designed to reflect a typical software router use case. Usually a router will encounter far more read requests (IP lookups) as compared to write requests (routing table updates) . To model this we consider pure reader workloads and workloads which are read intensive with a very low fraction of writes.

We used a realistic routing table derived from the routeviews.org database [1]. This table consists of 167,000 routes. We call this table the *167k table*. The input set was generated randomly using the 167k router configuration. A reader task consisted of performing lookups for 100,000 inputs from this input set. An updater task involves replacing routes for 1000 routes in the input set. Each of the reader and updater threads execute 128 such tasks in a single run of the test.

The benchmarks involving only readers were run on the RCU version, the reader-writer lock version and the vanilla version with no locks. When there were updaters involved in the workload, the benchmark was run on the RCU version and the reader-writer lock version. We cannot run the workload with writers on the vanilla version since it is not thread safe.

5.2.1 Pure Reader Workload

We first look at a workload consisting of only readers. The results are shown in Figure 11 and Table 7.

Workload	No Locks (s)	Reader-Writer Lock (s)	RCU (s)	$\frac{\text{Reader-Writer Lock}}{\text{No Locks}}$	$\frac{\text{RCU}}{\text{No Locks}}$
1 reader 0 writers	1.409	2.152	1.545	1.528	1.097
2 readers 0 writers	1.449	3.740	1.522	2.581	1.051
3 readers 0 writers	1.435	7.418	1.490	5.169	1.038
4 readers 0 writers	1.552	11.717	1.550	7.550	0.999
5 readers 0 writers	1.759	14.317	1.698	8.140	0.965
6 readers 0 writers	1.668	18.117	1.735	10.862	1.040
7 readers 0 writers	1.848	22.145	1.833	11.983	0.992
8 readers 0 writers	2.307	33.965	2.317	14.723	1.005

Table 7: Performance comparison over a workload with increasing number of readers using the 167k routing table. The first three columns show time in seconds. Smaller numbers are better.

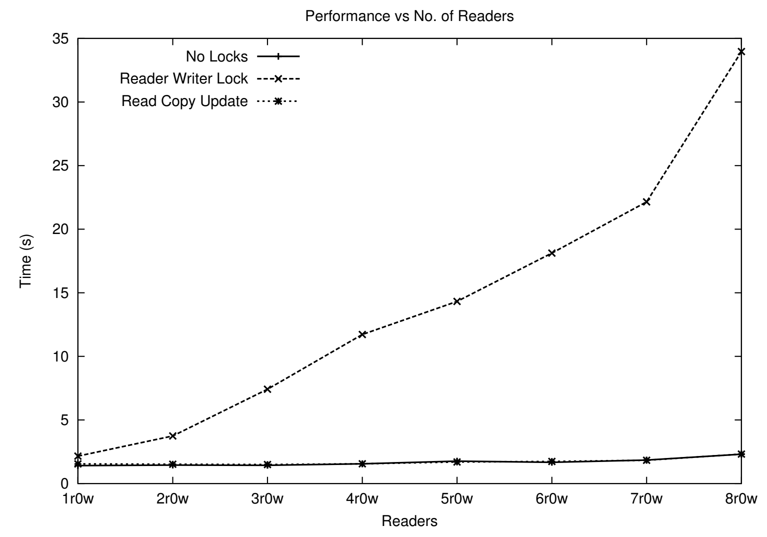


Figure 11: Performance of increasing number of readers with zero writers using the 167k routing table. Performance of RCU is very close to non-locking performance. Performance of the reader-writer lock degrades linearly.

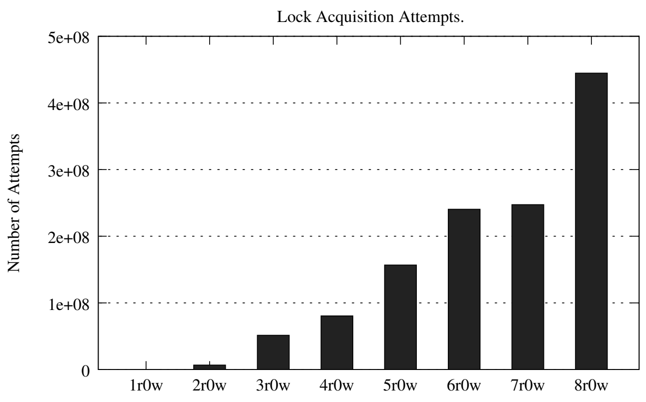


Figure 12: Number of lock acquisition attempts while using the reader-writer lock for a pure reader workload.

From Figure 11, we see that RCU scales much better than the reader-writer lock version. RCU is always within 1.1 times the vanilla version. Time taken by the reader-writer lock increases linearly and is up to 22 times slower than the version without locks.

These results validate our claim that RCU is wait-free and lock-free for readers. In the RCU version, readers do not create any synchronization overhead. However, readers in the reader-writer lock version access and update a shared lock variable before they access the routing table. The state of the shared variable needs to be updated through all cores. As the number of threads increases, the contention causes cache line bouncing and increased usage of the processor bus bandwidth. For the reader-writer lock, the number of lock acquisition attempts to enter the read side critical section is shown in Figure 12. The graph shows an increasing number of lock acquisition attempts with an increase in the number of threads. We can therefore attribute the linear increase in time to lock contention.

5.2.2 Read Intensive Workload With One Writer

Workload	Reader-Writer Lock (s)	RCU (s)	$\frac{\text{Reader-Writer Lock}}{\text{RCU}}$
1 reader 1 writer	2.268	1.550	1.463
2 readers 1 writer	3.022	1.575	1.919
3 readers 1 writer	7.683	1.595	4.817
4 readers 1 writer	10.388	1.712	6.066
5 readers 1 writer	13.960	1.785	7.821
6 readers 1 writer	17.980	1.917	9.377
7 readers 1 writer	22.443	2.380	9.430

Table 8: Performance comparison of increasing number of readers and one writer with the 167k routing table.

We also conducted a benchmark which consisted of one updater and an increasing number of readers. The results are shown in Figure 13 and Table 8.

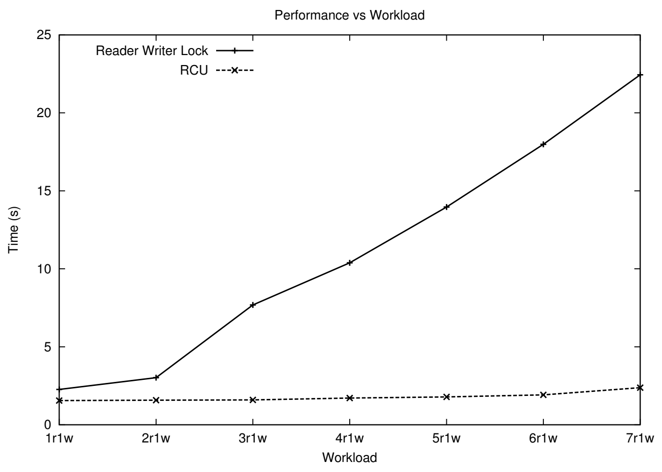


Figure 13: Performance of increasing number of readers and one writer with the 167k routing table.

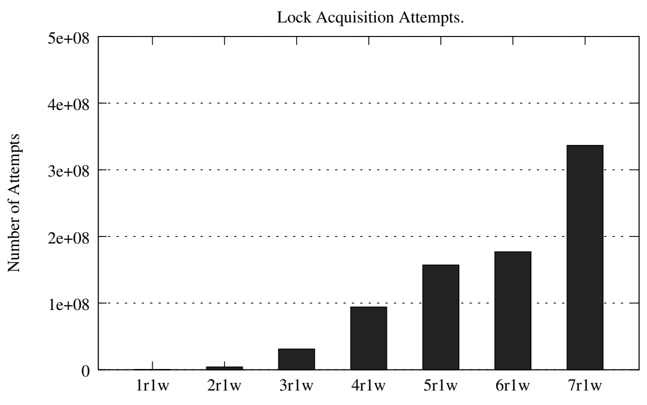


Figure 14: Number of lock acquisition attempts for workload with increasing number of readers and one writer while using a reader-writer lock.

We see from Figure 13 that RCU scales far better than the reader-writer lock. Time taken by the reader-writer lock increases linearly with the number of threads. The reader-writer lock version is over nine times slower than the RCU version for the workload consisting of 7 readers and 1 writer. Figure 14 shows the number of lock acquisition attempts while using the reader-writer lock on this workload. We can attribute the linear increase in time to the increasing lock contention. RCU does not face any lock contention—the updater acquires the lock every time it attempts to do so, since the workload has one updater. We can clearly see that RCU outperforms the reader-writer lock for a read intensive workload with some writers.

5.2.3 Write Intensive Workloads

We have seen that RCU outperforms the Reader-Writer lock for reader heavy workloads. For completeness, we look at some workloads which are not common for a router: writer heavy workloads. The performance of a workload with zero readers and increasing number of writers is shown in Figure 15 and Table 9. Here, we see that the Reader-Writer Lock outperforms RCU by a factor of 1.46. We attribute this to the increased memory footprint and increased overhead in management of the reclaim lists.

The performance of a workload consisting of one reader and increasing number of writers is shown in Figure 15 and Table 10. Although RCU does not scale, we see that it performs better than the Reader-Writer Lock.

It is clear that RCU is not as fabulous for write intensive workloads. These workloads are very uncommon for a typical router and we can accept this behavior.

Workload	Reader-Writer Lock (s)	RCU (s)	$\frac{\text{Reader-Writer Lock}}{\text{RCU}}$
0 readers 1 writer	0.180	0.182	0.986
0 readers 2 writers	0.203	0.225	0.900
0 readers 3 writers	0.230	0.295	0.780
0 readers 4 writers	0.258	0.362	0.710
0 readers 5 writers	0.307	0.435	0.707
0 readers 6 writers	0.352	0.502	0.701
0 readers 7 writers	0.407	0.588	0.694
0 readers 8 writers	0.453	0.660	0.686

Table 9: Performance comparison of increasing number of writers and zero readers using the 167k routing table.

Workload	Reader-Writer Lock (s)	RCU (s)	$\frac{\text{Reader-Writer Lock}}{\text{RCU}}$
1 reader 1 writer	2.260	1.540	1.468
1 reader 2 writers	2.335	1.593	1.466
1 reader 3 writers	2.397	1.677	1.429
1 reader 4 writers	2.467	1.962	1.257
1 reader 5 writers	2.547	2.078	1.226
1 reader 6 writers	2.605	2.408	1.082
1 reader 7 writers	2.663	2.310	1.153

Table 10: Performance comparison of increasing number of writers and one reader using the 167k routing table.

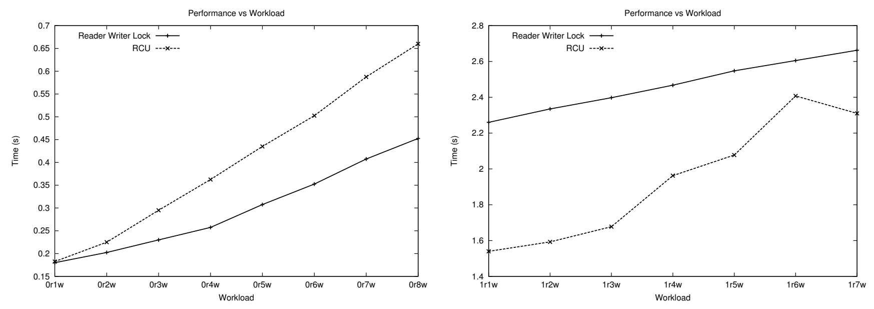


Figure 15: Write intensive workloads: Performance of increasing number of writers with no readers (left), and increasing number of writers with one reader (right). Both measurements used the 167k routing table.

Workload	No Locks (s)	Reader-Writer Lock (s)	RCU (s)	$\frac{\text{Reader-Writer Lock}}{\text{No Locks}}$	$\frac{\text{RCU}}{\text{No Locks}}$
1 reader 0 writers	0.355	0.555	0.372	1.563	1.049
2 readers 0 writers	0.352	3.692	0.378	10.475	1.071
3 readers 0 writers	0.362	6.495	0.482	17.917	1.331
4 readers 0 writers	0.378	9.477	0.403	25.106	1.066
5 readers 0 writers	0.420	14.607	0.475	34.780	1.131
6 readers 0 writers	0.445	18.410	0.490	41.371	1.101
7 readers 0 writers	0.453	22.775	0.500	50.331	1.105
8 readers 0 writers	0.465	26.960	0.512	57.978	1.102

Table 11: Performance comparison over a workload with increasing number of readers and zero writers.

5.3 Micro-benchmarks

The micro-benchmarks have tests which repeatedly access a small set of addresses in a very short time span. As with macro-benchmarks, we are interested mainly in read heavy workloads. We look at write heavy workloads for completeness.

Pure Reader Workload Here we consider a workload consisting of only readers. We look at how the reader-writer lock and RCU perform against the unmodified vanilla version. As we can see from Table 5.3 which is represented in Figure 16, the reader-writer lock is up to 50 times slower than a version without locks. The reader-writer lock does not scale with an increase in the number of threads. The RCU version scales with an increase in the number of threads. The RCU version is also within 1.5 times the performance of the version without any locks.

The Reader-Writer lock spends time acquiring a lock on reads while RCU does not have any additional overhead for readers. Any additional overhead of RCU over the vanilla version is due to the overhead in checking reclaim lists and detecting quiescent states.

Read Intensive Workloads We now compare the performance of RCU over the reader-writer lock in the presence of a writer. This workload reflects typical router usage. We refer the reader to Table 5.3 and Figure 17 for the results. RCU scales over an increase in the number of threads, however the reader-writer lock does not scale with an increase in the number of readers. This can be attributed to time spent waiting to acquire the lock before reading or writing in the Reader-Writer Lock version.

Write Intensive Workloads We now compare the performance of RCU with the reader-writer lock for a write intensive workload. The performance of a workload with one reader and increasing number of writers is shown in Table 5.3 and Figure 18. The performance of a workload with zero readers and increasing number of writers is shown in Table 5.3 and Figure 18. In both cases, the reader-writer lock performs better than RCU. We attribute this to additional overhead caused due to lock contention to free stale data in quiescent states. When there are more writers, there is more stale data. Since these benchmarks target a very small set of routes, the reclamation also has contention for the same memory locations.

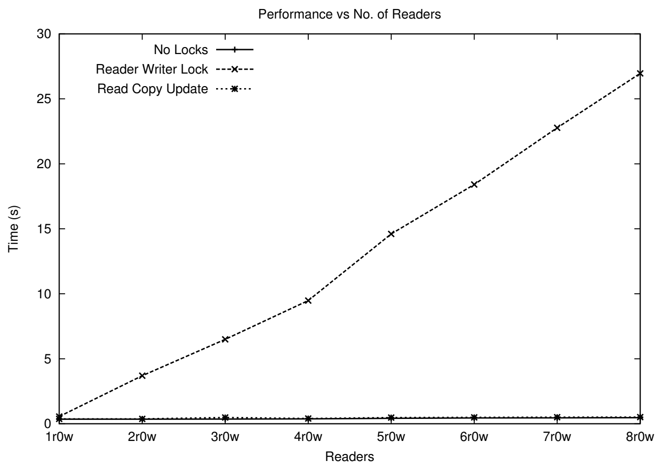


Figure 16: Performance comparison over a workload with increasing number of readers and zero writers.

Workload	Reader-Writer Lock (s)	RCU (s)	$\frac{\text{Reader-Writer Lock}}{\text{RCU}}$
1 reader 1 writer	0.863	0.598	1.444
2 readers 1 writer	4.253	0.630	6.750
3 readers 1 writer	6.805	0.700	9.721
4 readers 1 writer	10.850	0.748	14.515
5 readers 1 writer	14.590	0.740	19.716
6 readers 1 writer	18.733	0.873	21.470
7 readers 1 writer	23.285	0.865	26.919

Table 12: Performance comparison of increasing number of readers with one writer.

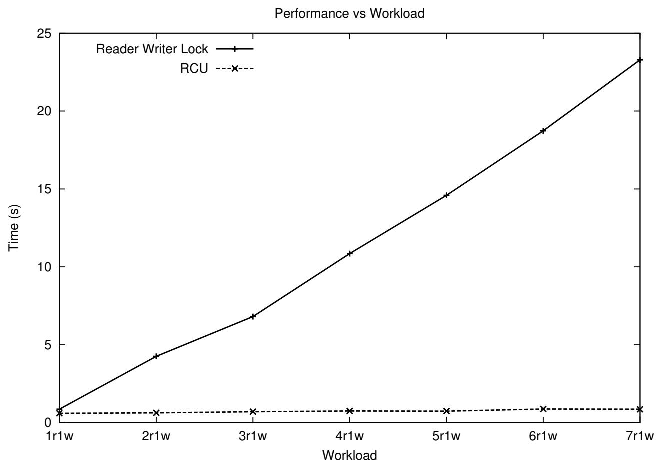


Figure 17: Performance comparison of increasing number of readers with one writer.

Workload	Reader-Writer Lock (s)	RCU (s)	$\frac{\text{Reader-Writer Lock}}{\text{RCU}}$
0 readers 1 writer	0.145	0.217	0.667
0 readers 2 writers	0.505	0.958	0.527
0 readers 3 writers	0.912	1.863	0.490
0 readers 4 writers	1.208	2.993	0.404
0 readers 5 writers	2.092	3.945	0.530
0 readers 6 writers	2.660	4.680	0.568
0 readers 7 writers	3.822	6.085	0.628
0 readers 8 writers	4.457	6.820	0.654

Table 13: Performance comparison of increasing number of writers with zero readers.

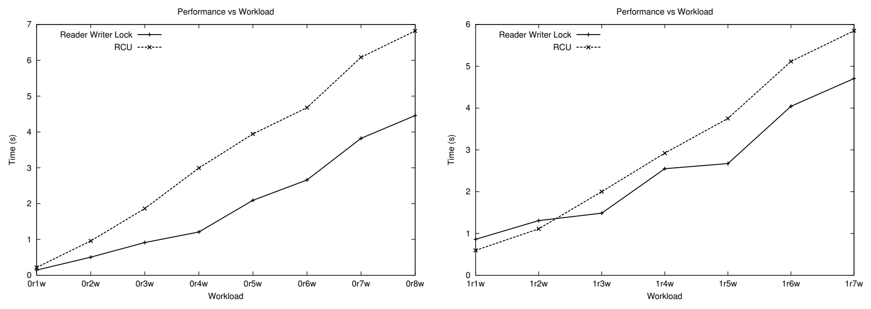


Figure 18: Write Intensive workloads: Performance comparison of increasing number of writers with zero readers (left), and one reader (right).

Workload	Reader-Writer Lock (s)	RCU (s)	$\frac{\text{Reader-Writer Lock}}{\text{RCU}}$
1 reader 1 writer	0.860	0.595	1.445
1 reader 2 writers	1.308	1.110	1.178
1 reader 3 writers	1.485	2.000	0.743
1 reader 4 writers	2.550	2.922	0.873
1 reader 5 writers	2.672	3.752	0.712
1 reader 6 writers	4.043	5.115	0.790
1 reader 7 writers	4.707	5.850	0.805

Table 14: Performance comparison of increasing number of writers with one reader.

6 Related Work

With over 2000 reported uses of the RCU API [18] in the Linux kernel in recent years, RCU is widely used in system software. It does not seem to be as popular in userlevel applications. Desnoyers et al. [7] attribute this to multiple factors, such as the constrained design imposed by previous userlevel RCU algorithms, heavy read-side overhead, etc. They propose and discuss several approaches to implementing RCU in applications. One such approach is the quiescent state based reclamation (QSBR) approach. A thread reaches a quiescent state when it releases references to mutable data structures. When each thread has reached a quiescent state, QSBR reclaims stale references. It has near zero read side overhead but has constraints on application design: when the thread lies in a quiescent state, it is required to call `rcu_quiescent_state()`. Another approach is what is referred to as *General Purpose RCU*. In contrast with QSBR, this approach does not impose any constraints on application design; it can even be used in implementing library functions and API. It has the highest read-side overhead as compared to the other RCU approaches. The overhead is due to memory barriers which are inserted before and after the read side critical sections. The third approach is the *Signal-Based RCU*. It avoids the high read-side overhead in the general-purpose approach by sending POSIX signal from the update side critical sections.

Click has natural quiescent states which allow us to implement QSBR. Each thread periodically calls a function which checks for a grace period in order to begin reclaiming stale data. Section 4.1.3 describes this mechanism in detail.

Hart et al. [12] compare several different memory reclamation schemes for RCU. Amongst these QSBR is again the scheme with the best performance, but is constrained by the requirement that the application has some natural quiescent states. The authors mention that the QSBR approach can be implemented through a fuzzy barrier. "A barrier protects access to some code which no thread should execute before all other threads finish some prior stage of computation." A fuzzy barrier is non-blocking, in contrast with a (non-fuzzy) barrier. If a thread cannot get past a fuzzy barrier, it skips the protected code and resumes execution. A (non-fuzzy) barrier would have the thread block until all threads have reached it. We describe how we implement the fuzzy barrier in Section 4.1. The epoch based reclamation (EBR) scheme is application independent, but has higher overhead than QSBR. EBR is similar to the general purpose RCU approach described by Hart et al. [7], in the sense that it is application agnostic. EBR has some read side overhead where QSBR has none. EBR uses epochs to implement a fuzzy barrier. Each thread updates local epoch numbers which are checked against a global epoch number to detect quiescent states. Although our approach is modeled on QSBR, it is similar to EBR for several reasons. Firstly, we use epochs to implement a fuzzy barrier. Secondly updaters and *readers* update a local epoch number as opposed to a pure QSBR approach where readers have zero overhead.

Dobrescu et al. [9] in *RouteBricks* propose an architecture which enables higher packet processing speeds by distributing router functionality across multiple servers and multiple cores within a server. Their approach involves load balancing through Valiant Load Balancing (VLB). Their method of achieving faster packet processing is at a coarser level of granularity when compared to our approach. Our approach complements RouteBricks, since a RouteBricks router could potentially achieve faster packet forwarding rates by using our RCU algorithms on its component bricks.

Chen and Morris [4] describe techniques for multiprocessor routing by optimizing scheduling, buffer and device management. Their technique uses a per CPU routing table which does not allow dynamic updates and deletions. Synchronization for other elements is achieved through reader-writer locks or through the atomic `xchgw` instruction. Our work can complement theirs as it provides a synchronization framework for all Click elements.

Multiple efforts have been made at exploiting parallelism in the packet-flow mechanism in a software router. Wu and Wolf [21] have designed a run-time system which distributes the allocation of processing tasks to processor cores. It also balances the workload and maps the it across multiple cores. This parallelism is done at a task-level granularity, and does not deal with synchronization of shared data structures used within a task. In that respect, our work complements that of Wu and Wolf [21], since it ensures that tasks are thread-safe.

Dobrescu et al. [8] have proposed a technique in which the packet-processing flow and server characteristics are taken as input to formulate an optimization problem. A compiler would then solve this problem and output machine code which will optimize for throughput.

7 Conclusion

The primary challenges in implementing RCU for Click were to identify quiescent states in a way which could be applied to all elements.

We have built an RCU framework for userlevel Click invlolving a mechanism to detect quiescent states and an API which can be used by userlevel Click elements.

We have verified through our benchmarks that this mechanism does indeed have a very low read side overhead (within 1.1 times non-locking performance). The RCU approach is also up to 27 times faster than a reader-writer lock on read intensive workloads. RCU's performance is not better than that of a reader-wrtier lock for write-intensive workloads. We find this behaviour acceptable since our workload is IP lookups, which is read-intensive.

8 Future Work

We have experimented with the RadixIPLookup Element in Click. DirectIPLookup and RangeIPLookup are some of the other fast routing table elements in Click. It would be useful to implement a RCU based synchronization these elements as well.

Our approach uses RCU for the vector in RadixIPLookup, however a coarse grained lock is acquired for updates to the radix tree. The size of the radix tree depends on the size of the routing table, which is typically very large. Thus implementing RCU based synchronization for the radix tree would be very useful.

A Source Code

A.1 Getting the Source Code

The source code is available in the *rcu_local_epoch* branch of the repository, `https://github.com/mvenk/click.git`. The microbenchmarks and macrobenchmarks can be found in the *race/microbenchmarks* and *race/macrobenchmarks* directories respectively.

A.2 Running Benchmarks

In order to run the benchmarks in multi-threaded mode, configure and install click like so:

```
$cd click_dir
$./configure --enable-user-multithread --enable-multithread=8
$ make -j 8
```

Note that we configure with *8*, since our experiments were run on an 8 core machine. The Makefile in the microbenchmarks and macrobenchmarks directories contain a brief description of the tests. To run a macrobenchmark with varying number of readers and one writer, do the following:

```
$cd race/macrobenchmarks
$make rcu_vr_0w
```

In order to compare against the reader-writer lock version or the non-locking version, run the same tests in the *rcu_master_tests* branch. All tests are configured for a machine with 8 cores. In order to generate tests for N number of cores, use the *gen_benchmarks.sh* script provided in the *race/macrobenchmarks* directory.

References

[1] The routeviews project. `http://archive.routeviews.org/`.

[2] Katerina J. Argyraki, Salman Baset, Byung-Gon Chun, Kevin R. Fall, Gianluca Iannaccone, Allan Knies, Eddie Kohler, Maziar Manesh, Sergiu Nedevschi, and Sylvia Ratnasamy. Can software routers scale? In Jennifer Rexford, Jacobus E. van der Merwe, and T. V. Lakshman, editors, *Proceedings of the ACM SIGCOMM 2008 Workshop on Programmable Routers for Extensible Services of Tomorrow, PRESTO 2008, Seattle, WA, USA, August 22, 2008*, pages 21–26. ACM, 2008.

[3] AT&T". At&t completes next-generation IP/MPLS backbone network, world's largest deployment of 40-gigabit connectivity. `http://www.att.com/gen/press-room?cdvn=news&newsarticleid=26230&pid=4800`.

[4] Benjie Chen and Robert Morris. Flexible control of parallelism in a multiprocessor PC router, 2001.

[5] Tzi cker Chiueh and Prashant Pradhan. High-performance IP routing table lookup using cpu caching.

[6] L. Zhang D. Meyer and K. Fall. Report from the IAB workshop on Routing and Addressing. RFC 4984, 2007.

[7] Mathieu Desnoyers, Paul E. McKenney, Alan Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23:375–382, 2012.

[8] Mihai Dobrescu, Katerina Argyraki, Gianluca Iannaccone, Maziar Manesh, and Sylvia Ratnasamy. Controlling parallelism in a multicore software router. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 2:1–2:6, New York, NY, USA, 2010. ACM.

[9] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM.

[10] Dinakar Guniguntala, Paul E. Mckenney, Josh Triplett, and Jonathan Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with linux. *IBM Systems Journal*, 47(2):221–236, 2008.

[11] Pankaj Gupta, Steven Lin, and Nick McKeown. Routing lookups in hardware at memory access speeds. pages 1240–1247, 1998.

[12] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67:1270–1285, December 2007.

[13] Zhuo Huang, David Lin, Jih-Kwon Peir, Shigang Chen, and S. M. Iftekharul Alam. Fast routing table lookup based on deterministic multi-hashing. In *Proceedings of the The 18th IEEE International Conference on Network Protocols*, ICNP '10, pages 31–40, Washington, DC, USA, 2010. IEEE Computer Society.

[14] Eddie Kohler. IpRouteTable element documentation. `http://read.cs.ucla.edu/click/elements/iproutetable`.

[15] Eddie Kohler. RadixIPLookup documentation. `http://read.cs.ucla.edu/click/elements/radixiplookup`.

[16] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18:263–297, August 2000.

[17] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[18] Paul E. McKenney and Jonathan Walpole. Introducing technology into the Linux kernel: a case study. *SIGOPS Oper. Syst. Rev.*, 42:4–17, July 2008.

[19] T. Narten, E. Nordmark, and W. Simpson. Neighbor Discovery for IP Version 6 (IPv6). RFC 2461 (Draft Standard), December 1998. Obsoleted by RFC 4861, updated by RFC 4311.

[20] Werner Vogels. Eventually consistent. *Queue*, 6:14–19, October 2008.

[21] Qiang Wu and Tilman Wolf. On runtime management in multi-core packet processing systems. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 69–78, New York, NY, USA, 2008. ACM.