Read Copy Update Framework for Userlevel Click

Madhuri Venkatesh
UCLA
madhuri@cs.ucla.edu

Supervised By:
Eddie Kohler
UCLA/Harvard
kohler@cs.ucla.edu

January 8, 2012

1 Abstract

High performance routing lookups are critically important for packet forwarding. With multi-core systems becoming more popular and more prevalent, leveraging them is key to improving performance. However, obtaining a correct, scalable and fast thread-safe solution remains challenging.

We have described an approach for providing thread-safety using *Read Copy Update*. We have verified that we incur low read side overhead and that our approach performs up to 80% better than reader-writer locks on read-intensive workloads.

2 Introduction

The ever-increasing volume of network traffic over the internet demands highly efficient routing lookups. Link rates today exceed 40 Gbps, commodity network cards operate at over 15 Gbps but software routers struggle to comfortably scale beyond 10 Gbps.

Commodity machines with SMP multi-core architectures are growing in popularity. A software modular router such as Click 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. A solution using locks also invariably involves the use of variables to synchronize access. These variables need to be accessed simultaneously by other processes or threads causing *cache line bouncing*. This refers to the overhead of communicating the shared value across multiple caches which is very expensive. The larger the number of threads, the larger is the frequency of access of synchronization variables and the larger is the cost of synchronization. The lock based approach becomes less scalable and hence less attractive.

It becomes essential to carefully evaluate our context and design a multi-core, multi-threaded solution which is highly efficient. Our context involves a read intensive workload comprising of 90% readers and less than 10% updaters. *Read Copy Update* [3] scales efficiently on read intensive workloads and we seek to implement a Read Copy Update framework for the Click modular router. We compare the performance of our approach against a traditional lock-based approach using Click on a commodity machine.

The rest of the document is organized as follows. Section 3 gives the reader a background on the userlevel Click elements we are concerned with. Section 5 explains the main problems which exist in the current implementation. Section 4 describes our so-

lutions to the problem along with the implementation details. Sections 6 and 7 describe the performance hypotheses and analysis.

3 Related Work

Routebricks [2] proposes 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 with our approach.

Multiple efforts have been made at exploiting parallelism in the packet-flow mechanism in a software router. Wu and Wolf [5] 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. It does not look at synchronization of objects being used by the task. In our work we attempt to allow the task itself to be thread-safe so that it can be run on multiple cores.

Dobrescu et al. [1] 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. Our work could complement [5], since by creating a thread-safe solution, we eliminate dependencies within the elements. This means that we do not have to worry about the order of execution of tasks which share data.

4 Background

We seek to find a scalable way to run userlevel Click elements in a multithreaded environment. Our approach was to try our solutions on the *RadixIPLookup* element. We then extend our solution to other elements in Click. This section presents a brief overview of the RadixIPLookup element.

4.1 An overview of RadixIPLookup

The *RadixIPLookup* element [4] is a class which is used to perform IP lookups. It uses a radix trie to perform the lookups and a vector to store the routes. This structure is explained in further detail.

4.1.1 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 when the last level is fully occupied. Figure 1 shows the tree pictorially.

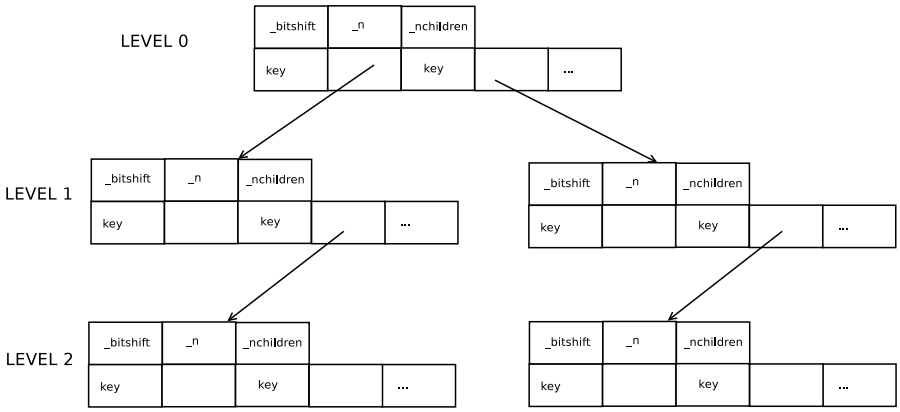

Figure 1: The Radix Tree

Each level consists of radix nodes. The structure of the radix node is shown in Listing 1. *_n* represents the number of radix nodes at that level. *_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

```
1  class Radix {
2    private:
3
4      int _bitshift;
5      int _n;
6      int _nchildren;
7      struct Child {
8          int key;
9          Radix *child;
10     } _children[0];
11 ...
12 ...
13 }
```

Listing 1: The Radix Class

into the routing table.

Lookups Routing Table Lookups for the Internet Protocol use a longest prefix match algorithm matching a candidate route with the highest subnet mask. The node at the top most level stores keys which need to match upto 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 a 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.

```
1   static inline int lookup(const Radix *r, int cur, uint32_t addr) {
2     while (r) {
3       int i1 = (addr >> r->_bitshift) & (r->_n - 1);
4       const Child &c = r->_children[i1];
5       if (c.key)
6       cur = c.key;
7       r = c.child;
8     }
9     return cur;
10  }
```

Listing 2: The lookup function

4.1.2 The Vector class

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*. The names of the first four fields are self-explanatory, Extra is used to recycle previously used but currently unused space within the vector. If the index in the vector is in use Extra is set to -1. If an entry in the vector is deleted, the index of that entry is added to a free list.

The use of *extra* is illustrated with the help of an example. Table 4.1.2 represents the vector at some point in time. The indices 2, 4 and 5 in the vector are currently being used (hence their *_vfree* is set to -1). The indices 0, 1 and 3 were in use earlier and have been deleted. They are part of the freelist. The *_vfree* points to the first entry in the freelist which will be used next. Briefly, `freelist::{0 -> 3 -> 1,_vfree= 0}`. Let us say a new entry is added to the vector. So *_vfree* is reused, and the freelist is updated. Now, `freelist::{ 3 -> 1, _vfree =3}`. Following another update, the index 3 will be used and the `freelist::{1,_vfree=1}`.
If the index 2 is now freed, `freelist::{2 -> 1,_vfree = 2}`.

Dynamic resize As is the case with most Vector classes, the Vector class in click has an _push_back() function which adds an entry to the vector.

If _n exceeds _capacity, a call to reserve_and_push_back() is made. This dynamically increases the size of the vector. The code for push_back and reserve_and_push_back() is given in the code-listing.

index	addr	mask	port	gw	extra
0	0.0.0.1	255.255.255.0	1	22.34.198.3	3
1	1.1.0.1	255.255.255.0	1	1.34.198.3	-1
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	-1

Table 1: Vector with routes

5 The Problem

Our goal is to improve lookup performance by allowing for multithread access to Click elements. In this section we look at problems in the existing code and the key factors which hinder safe multithreaded access.

The section is divided into unsafe access caused by multiple updaters (Updater-Updater Conflicts) and unsafe access caused by concurrent readers and updaters (Reader-Updater Conflicts). Concurrent updaters breed race conditions which might lead to an inconsistency in the data structures being used. A reader racing with an updater might read stale or invalid data.

5.1 Updater-Updater Conflicts

We now deal with race-conditions and conflicts arising due to multiple updaters running concurrently on the same instance of 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 tree times the size of a radix node.

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

```
1  int
2  RadixIPLookup::Radix::change(uint32_t addr, uint32_t mask,
3                                          int key, bool set)
4  {
5    int i1 = (addr >> _bitshift) & (_n - 1);
6    // check if change only affects children
7    if (mask & ((1U << _bitshift) - 1)) {
8      if (!_children[i1].child
9      && (_children[i1].child = make_radix(_bitshift - 4, 16)))
10     ++_nchildren;
11     if (_children[i1].child)
12     return _children[i1].child->change(addr, mask, key, set);
```

```
13        else
14        return 0;
15      }
16      .......
17      .......
18      .......
19    }
```

Listing 3: Race in Change

The change() code 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. Looking at line number 9, we see that if there are multiple threads executing change() there is a race for the assignment of the radix node. Multiple threads may call *make_radix()*, however only one pointer which is returned by *make_radix()* is assigned. This behavior can lead to memory leaks.

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 which includes the size and free-list. Since the vector is resized dynamically, we could lose updates and have memory-leaks if reserve() is called concurrently.

The conflicts described above are better understood with the help of code. We will attempt to explain the conflicts using Listing 4.

```
1   int
2   RadixIPLookup::add_route(const IPRoute &route,
3                            bool set,
4                            IPRoute *old_route,
5                            ErrorHandler *)
6   {
7     int found = (_vfree < 0 ? _v.size() : _vfree), last_key;
8     if (route.mask) {
9       uint32_t addr = ntohl(route.addr.addr());
10      uint32_t mask = ntohl(route.mask.addr());
11      last_key = _radix->change(addr, mask, found + 1, set);
12    } else {
13      last_key = _default_key;
14      if (!last_key || set)
15      _default_key = found + 1;
16    }
17    if (last_key && old_route)
18    *old_route = _v[last_key - 1];
19    if (last_key && !set)
20    return -EEXIST;
21    if (found == _v.size())
22    _v.push_back(route);
23    else {
24      _vfree = _v[found].extra;
25      _v[found] = route;
26    }
```

```
27      _v[found].extra = -1;
28      if (last_key) {
29        _v[last_key - 1].extra = _vfree;
30        _vfree = last_key - 1;
31      }
32      return 0;
33    }
```

Listing 4: The add_route function

We consider two cases, one in which the concurrently executing threads retrieve an index freelist, the other in which the concurrently executing threads make a call to *push_back()* on the vector.

Case 1: Concurrent updaters use the freelist Let us assume that *Thread 1 calls add_route(X,1,NULL,NULL), X.addr =A, Y.mask = M*. 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.

Thread 2 calls *add_route(Y,1,NULL,NULL) Y.addr = A, Y.mask =M* .
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 8 first. This updates the key in the radix tree to be the value *found*.

Updater 2 executes line 8. The value of the key is set to *found*.
Updater 1 executes the remaining code incuding line 26 which sets *_v[found].extra* to -1 (which means that the index is in use).
Updater 2 now sets *_vfree* to *_v[found].extra* which is -1. If *_vfree* is -1, it means that there are no more free keys in the vector, which may not be true. This brings the vector into an inconsistent state.

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.

```
1   template <class T> inline void
2   Vector<T>::push_back(const T& x)
3   {
4     if (_n < _capacity) {
5       new(velt(_n)) T(x);
6       ++_n;
7     } else
8     reserve_and_push_back(RESERVE_GROW, &x);
9   }
```

Listing 5: The push_back function

We first consider the case when there is enough space in the vector: the expression $\_n < \_capacity$ evaluates to true. Updater1 and Updater2 use the same value of *_n*. The updater which wins the race installs its key at position i, and the other update is lost. If the aforementioned expresssion evaluates to false, we can have a different race which could lead to memory leaks. It is better understood with the help of the *reserve_and_push_back()* function shown in Listing 6.

```
1   Vector<T>::reserve_and_push_back(size_type want, const T *push_x)
2   {
3     if (want < 0)
4     want = (_capacity > 0 ? _capacity * 2 : 4);
5     if (want > _capacity) {
6       T *new_l = (T *) CLICK_LALLOC(sizeof(T) * want);
7       if (!new_l)
8       return false;
9       for (size_type i = 0; i < _n; i++) {
10        new(velt(new_l, i)) T(_l[i]);
11        _l[i].~T();
12      }
13      CLICK_LFREE(_l, sizeof(T) * _capacity);
14      _l = new_l;
15      _capacity = want;
16    }
17    if (unlikely(push_x))
18    push_back(*push_x);
19    return true;
20  }
```

Listing 6: reserve_and_push_back()

Assume that both Updater1 and Updater2 execute code concurrently and that each of them have a reference to a new array which they have created in line 7: *new_l*.
Some of these races could bring the vector to an inconsistent state or cause memory leaks. For example: If both the updaters call CLICK_LFREE() in line 14, we could have a segmentation fault. The assignment of *new_l* in line 16 is also a race. This could 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.

Verifying conflicts in Click An approach similar to the one described in the Reader-Updater Conflicts section was followed here. We created a Click elements called BashRadixIPLookup and PoundRadixIPLookup which repeatedly add and delete the same route having the same address but different values of the port. Both BashRadixIPLookup and PoundRadixIPLookup run concurrently on the same instance of RadixIPLookup. This concurrency is ensured by using the *StaticThreadSched()* element in the script.

```
BashRadixIPLookup::repeat 10,000 times {
  add_route (A,X);
```

```
  delete_route (A,X);
}
PoundRadixIPLookup::repeat 10,000 times {
  add_route (B, Y);
  delete_route (B,Y);
}
```

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

5.2 Reader-Updater Conflicts

The conflicts in the RadixIPLookup element occur primarily in two data structures: vector used to store the routes, and the radix-tree which stores a key indexing into the vector. The sections below describe the conflicts which occur in each of these data structures.

Radix Tree: This section describes some of the reader-updater conflicts which exist and why they are not a source for concern. We take a look at a race which might arise and explain why it is not a serious problem.

We now look at deletion of nodes within a radix tree. All pointers to the radix node are freed during a clean up phase of RadixIPLookup before it is destroyed. When a route is removed, the radix node corresponding to the route is not freed immediately. Instead all the radix nodes in the tree are deleted together at the end. This takes place in the call to free_radix(). free_radix() is called on an instance of RadixIPLookup just before the instance is destroyed.
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, the reader performs a lookup on the very route X.

We have two threads one running: *Updater::* {*remove Route X*} and the other running: *Reader::*{*read Route A*}. The reader might receive stale information for Route X. This happens if the reader acquires a reference to the radix node containing Route X before the pointer is removed from the tree (before the call to change(), line 24 in Listing 7). The race is illustrated with the help of the sequence diagram in 2. However, readers which execute after the updater has finished will not recieve stale data for Route X.

```
1  int
2  RadixIPLookup::remove_route(const IPRoute& route,
3                              IPRoute* old_route,
4                              ErrorHandler*)
5  {
6    int last_key;
7    if (route.mask) {
8      uint32_t addr = ntohl(route.addr.addr());
```

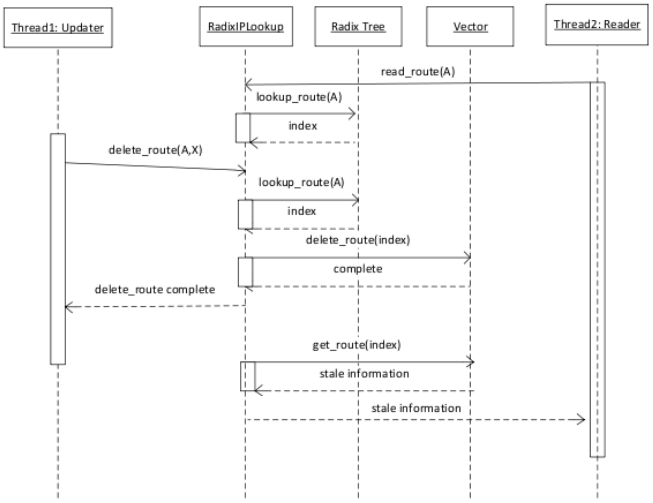

Figure 2: Race condition with a concurrent reader and updater

```
9       uint32_t mask = ntohl(route.mask.addr());
10      // NB: this will never actually make changes
11      last_key = _radix->change(addr, mask, 0, false);
12    } else
13    last_key = _default_key;
14    if (last_key && old_route)
15    *old_route = _v[last_key - 1];
16    if (!last_key || !route.match(_v[last_key - 1]))
17    return -ENOENT;
18    _v[last_key - 1].extra = _vfree;
19    _vfree = last_key - 1;
20    if (route.mask) {
21      uint32_t addr = ntohl(route.addr.addr());
22      uint32_t mask = ntohl(route.mask.addr());
23      (void) _radix->change(addr, mask, 0, true);
24    } else
25    _default_key = 0;
26    return 0;
27  }
```

Listing 7: The remove_route() function

The occurrence of the race mentioned above is rather rare: firstly a read intensive, infrequent update workload is expected but the aforementioned race is more likely in an update intensive workload. Secondly reads are fast compared to updates, so the possibility that a read operation starts before an update and ends after an update is

quite small. That being said, we assume that the consequences of the race illustrated in the example above are acceptable. This is justified by the fact that the reader received a stale route - it was once valid route for the given address, and the price is the price of one wrong lookup.

Vector: The library files in Click include a generic dynamically resized Vector class. This class and its use in RadixIPLookup has been explained in the Background section. The conflicts are explained in the context of the add_route() function which adds a route to the radix tree. The code for add_route is given in Listing 8. The way an index is obtained for insertion is summarized in the pseudocode below.

```
1  if(free_list is not empty)
2  {
3    index = free_list;
4    free_list = free_list -> next;
5    _v[free_list.index] = val;
6  }
7  else
8  _v.push_back(val);
```

Listing 8: Pseudocode for acquiring an index in add_route

We can observe at least two inadmissible conflicts engendered by races in the vector:

1. 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.

2. 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 occasioning a segmentation fault.

Verifying conflicts in Click The first conflict mentioned above was verified by running Click scripts. New elements were created and run in multi-threaded mode.The *StaticThreadSched()* element was used to specify the thread on which the element must be run. Listing 9 facilitates a clearer picture of the use of *StaticThreadSched()* for this case.

```
1  Idle
2   -> r :: RadixIPLookup106(
3                  0.0.0.0/0   8.1.1.1 0,
4                  )
5   -> Idle;
6
7  reader :: ReadRadixIPLookup106(r);
8  writer0:: BashRadixIPLookup106(r);
9  writer1 :: PoundRadixIPLookup106(r);
10
11 StaticThreadSched(
12         reader 0,
13         writer0 1,
14         writer1  2);
```

```
15
16 DriverManager(stop);
```

Listing 9: Click script for verifying reader-updater conflicts

We created a Click elements called BashRadixIPLookup and PoundRadixIPLookup which repeatedly add and delete a route having the for the same IP address but different port values.

```
BashRadixIPLookup::repeat 10,000 times {
  add_route (A,X);
  delete_route (A,X);
}
PoundRadixIPLookup::repeat 10,000 times {
  add_route (B, Y);
  delete_route (B,Y);
}
ReadRadix:: repeat 10,000 times {
  port = Read (A);
}
```

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. The parameters for *Add route* mentioned here are not the same as add_route() used in *RadixIPLookup*. The pseudocode presented above is simplified to illustrate the motive behind our method of verifying conflicts.
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 3

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.

6 The Solution

The race conditions described in the previous sections occur due to threads accessing shared data structures. In order to provide a thread-safe solution, several solutions were tried and their performance was compared. This section describes the different approaches tried in order to solve the problem. Coarse grained locking, fine-grained locking and Read Copy Update are the different methods we have tried. The Read Copy Update approach is the focus of this document, and a justification of its working and the implementation details are exposed in section 6.3

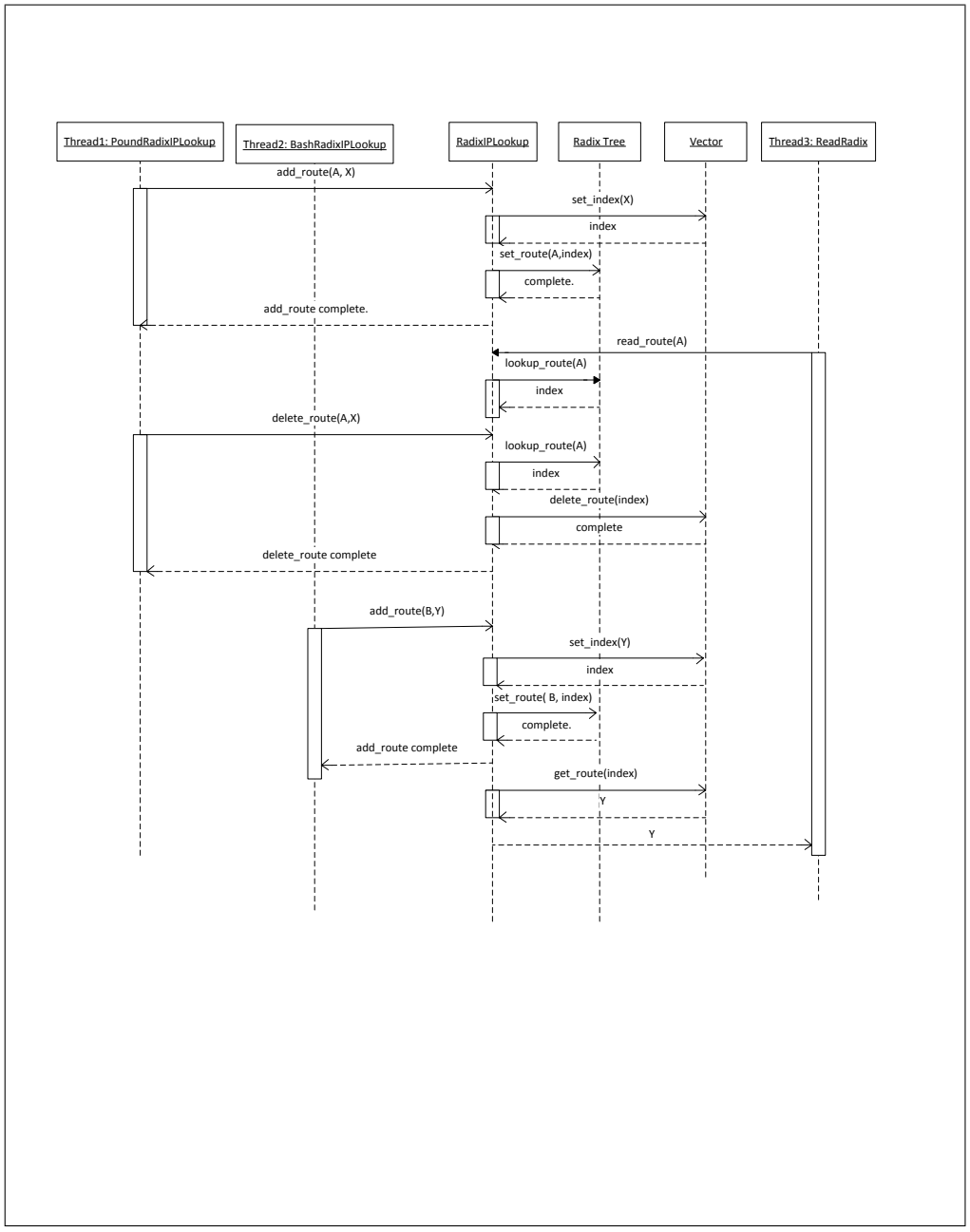

Figure 3: Race condition with a concurrent reader and updater

6.1 Coarse-grained locking

Locking solves our problems since it prevents two threads from accessing shared data structures at the same time. As our problem is in the context of a read intensive workload, a reader-writer lock was chosen.
The lock is acquired around the entire update and read section to ensure correctness and see that the conflicts do not occur. The *add_route()* function using the reader-writer lock is shown in Listing 11. This function behaves as a writer since it updates shared data structures. The *lookup_route()* function is a reader since it only reads shared data structures but does not modify them. The version using reader-writer lock is shown in Listing 10.

```
1  int
2  RadixIPLookup107::lookup_route(IPAddress addr, IPAddress &gw) const
3  {
4      _lock.acquire_read();
5      int key = Radix::lookup(_radix, _default_key, ntohl(addr.addr()));
6      int port = -1;
7      gw = 0;
8      if (key) {
9          gw = _v[key - 1].gw;
10         port =  _v[key - 1].port;
11     }
12     _lock.release_read();
13     return port;
14 }
```

Listing 10: Reader-writer lock usage in lookup_route()

```
1  int
2  RadixIPLookup107::add_route(const IPRoute &route,
3                              bool set,
4                              IPRoute *old_route,
5                              ErrorHandler *)
6  {
7    _lock.acquire_write();
8    // ...
9    // add_route() code
10   // ...
11   _lock.release_write();
12   return 0;
13 }
```

Listing 11: Reader-writer lock usage in add_route()

6.2 Fine-grained locking

The reader-writer lock approach was improved with fine-grained locking in the updates. Every update in RadixIPLookup involves changes to the radix tree and changes to the underlying vector that it uses. These two changes are coupled together in the sense that one update depends on the outcome of the other. The code was restructured in add_route() and remove_route() in order to make fine grained locking possible.

Changes to add_route The changes made to add_route for fine grained locking are described. add_route is used to install a new route into the radix tree. This involves two steps:

1. Adding the route into the vector, and obtaining a key into the index where the new route is installed.
2. Updating the radix tree to reflect the key obtained in 1.

The changes to the vector and changes to the radix tree are interdependent as seen in the *add_route()* function in Listing 5. add_route() was decoupled to use the following steps:

1. Obtain an index to install the new route.
2. Update the radix tree with the key for the new route (creating new branches if necessary).
3. Update the freelist to reflect changes if it had been used in step 1.
4. Delete the old key in the vector if it was found for that route in step 2.

After we have decoupled it in this way, we can create a thread safe implementation if each of the steps is atomic. The implementation involved locking and changes to the vector. The code for the modified version of add_route can be found in Listing 12.

```
1  int
2  RadixIPLookup106::add_route(const IPRoute &route,
3                              bool set,
4                              IPRoute *old_route,
5                              ErrorHandler *)
6  {
7    int last_key;
8    // we optimistically push back the route onto
9    // the vector and don't do any free list management
10   int found = insert_into_v(route);
11   if (route.mask) {
12     uint32_t addr = ntohl(route.addr.addr());
13     uint32_t mask = ntohl(route.mask.addr());
14     _rlock.acquire();
15     last_key = _radix->change(addr, mask, found + 1, set);
16     _rlock.release();
17   } else {
18     last_key = _default_key;
19     if (!last_key || set)
20     _default_key = found + 1;
21   }
22   if (last_key) {
23     if(old_route) {
24       _vlock.acquire();
25       *old_route = _v[last_key - 1];
26       _vlock.release();
27     }
28     if(set)
29     remove_from_v(last_key);
```

```
30        else {
31          remove_from_v(found);
32          return -EEXIST;
33        }
34      }
35      return 0;
36    }
```

Listing 12: Fine-grained add_route()

Changes to remove_route: The issues with *add_route()* also exist with *remove_route()*. This is because *remove_route()* also involves changes to the radix tree and changes to the underlying vector it uses in a similar manner. The restructured code for fine-grained locking works as shown below:

1. Search for the route in the radix tree and store it in last_key.
2. If there was no route found, return with the appropriate error code.
3. If there was a route found, update the radix tree to reflect the change.
4. Update the vector and its free-list.

Changes to the Vector: The vector can be resized dynamically with a call to *reserve()*. If multiple updaters concurrently call reserve(), there can be lost updates or segmentation faults due to the race conditions. Our goal was to modify the vector such that it is thread safe and wait-free for readers.
We created a two level vector: the first level has an array of pointers to other vectors and the second level stores the values. This introduces another level of indirection. An element whose index appears to be *x* is actually located at *_v[pdx][idx]*, where *pdx = x/BUCKETSIZE* and *idx = x % BUCKETSIZE*. This is illustrated in Figure 6.2. Now, during a *push_back()* operation if there is enough space in the vector, the value will be added in one of the vectors. If there is no space, a new vector is created and a pointer to it is stored in the bucket array. This is safe for readers, since a reference acquired by a reader is always valid (assuming that the indices are not reused).
Since updaters could add new vectors to the bucket array, we still need to acquire a lock on updates. We have tried to minimize locking and we use compare-and-swap instructions to increase the *_capacity* variable in the vector. A lock is acquired in *reserve()* in which the array is dynamically resized. This is done to prevent a race amongst concurrent updaters from executing *push_back()* in order to place new a vector into the bucket array. The code for *reserve()* which includes this modification is shown in Listing 13.

```
1 template <class T> inline bool
2 BucketArray<T>::reserve() {
3   _lock.acquire();
4   if(_nelems >= capacity()) {
5     T ** new_l = (T **) new unsigned char[\
6       sizeof(T*) * (_npointers + 1)];
7     if(!new_l) {
```

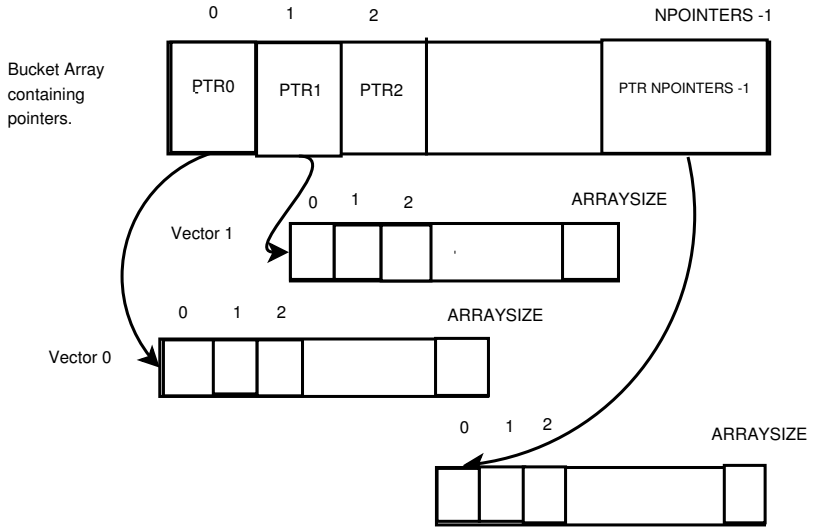

Figure 4: The bucket vector

```
8         _lock.release();
9         return false;
10      }
11
12      new_l[_npointers] = (T *) CLICK_LALLOC(sizeof(T) * ARRAY_SIZE);
13
14      if(!new_l[_npointers]) {
15        _lock.release();
16        return false;
17      }
18
19      memcpy(new_l, _l, sizeof(T**) * _npointers);
20      _reclaim_later.push_back((void*)_l);
21      //_reclaimhook.schedule();
22
23      _l = new_l;
24      _npointers++;
25
26    }
27    _lock.release();
28    return true;
29  }
```

Listing 13: Locking in reserve()

6.3 Using Read Copy Update in Click

Coarse and fine-grained locking solve the problems we have identified, but the locking system is still over 13 times slower in a pure reader workload consisting of four readers. Our goal is to develop a solution with no read-side overhead, so we turn to

Read-Copy-Update [3].

Using RCU requires that we identify *grace periods*: a state in which each thread has undergone at least one *quiescent state*. A quiescent state is a state where a thread does not have any references to mutable data structures. This could be an idle-loop or a state where the thread is running code which does not access any shared data.

One of the main challenges in implementing RCU for Click was identifying quiescent states. We solved this problem in RadixIPLookup using a timer expiry as a grace period. We then describe how we can find quiescent states in the Click scheduling loop. This is a quiescent state for any userlevel element. The next challenge was to provide an API which allows any userlevel element in Click to use RCU.

6.4 RCU using Timers

Assumptions: The lookups on RadixIPLookup take place within about 1000 ns. It is somewhat safe to assume that a reader would have finished within 500 ms. We use this time-interval as a grace period or a quiescent state to free data-structures periodically.

Implementation: Click provides a Timer class which runs call-back functions at specified time-intervals. We use this to schedule a callback which reclaims freed data. We maintain two lists: a *reclaim_now* list and a *reclaim_later* list. When an updater attempts to free an index we do not reclaim the index right away. Instead we add it to the *reclaim_later* list. In the timer callback function, we do the following
(a)reclaim everything in the reclaim_now list
(b)Swap the reclaim_now list and the reclaim_later list.
RCU's main benefit is that we never block readers. This means that any mutable data structures being read must always be in a consistent state. The original vector was dynamically resized and there could be cases where there is allocated but uninitialized memory. To solve this problem, the modified version used the bucket array described earlier with memory barriers.

6.5 RCU using Click Scheduling Loop

The timer expiry as a quiescent state described in Subsection 6.4 appears to work well for RadixIPLookup since each lookup is almost guaranteed to finish within 500 ms. This is not true of other elements in Click and it may not be possible to determine if and when a reader terminates. It becomes necessary to create a generic framework so that we can detect quiescent states for all other userlevel elements without having to know how long it takes for a reader task to complete. In this section, we look at modifying the core scheduling loop of Click in order to detect quiescent states.

The Click RouterThread driver loop Click allows the user to supply the maximum number of threads it can use. Click has a class called *RouterThread* which is responsible for running tasks scheduled by each of the elements in Click. 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 tasks, control periodically returns to the driver loop. A highly abridged version of the driver loop is shown in listing 14.

```
1  void RouterThread::driver()
2  {
3    // initialization
4    driver_loop:
5    run_tasks();
6    // perform scheduling tasks etc
7    // run timers
8    // if driver has been stopped, exit the loop
9    goto driver_loop;
10 }
```

Listing 14: Pseudocode for the driver loop

The instances of RouterThread are created by an instance of the Master class. Each thread is always in one of the following states:

```
S_PAUSED: Thread is paused.
S_BLOCKED: Thread is blocked.
S_TIMERWAIT : The thread is waiting for a timer.
S_LOCKSELECT:
S_LOCKTASKS:
S_RUNTASK: Thread is running a task.
S_RUNTIMER: Thread is running a timer.
S_RUNSIGNAL:
S_RUNPENDING:
S_RUNSELECT:
```

Local Epochs and the search for quiescent-states An epoch is the period during which the *RouterThread* performs some activity: running timers, running tasks etc. Each instance of *RouterThread* maintains an epoch number hereafter referred to as "local epoch number". The local epoch number is incremented every time control returns to the driver loop after finishing some activity. We also maintain a global epoch number. The global epoch number is incremented only when the local epoch numbers of all RouterThreads have been incremented. This is illustrated in the Figure 6.5. Every alternate global epoch we can free accumulated stale data. Specifically, we can safely free data accumulated by epoch X in epoch $X+2$. Note that we cannot free stale data every time the global epoch number increases (i.e. at $X+1$) since a reader may still have a reference to some stale data. For example, in Figure 6.5, we cannot free all stale data accumulated in $GE = 0$ in $GE = 1$ since thread 3 might still have a reference to something which has been updated by Thread 2. The states which might be sharing references are highlighted using the same color.

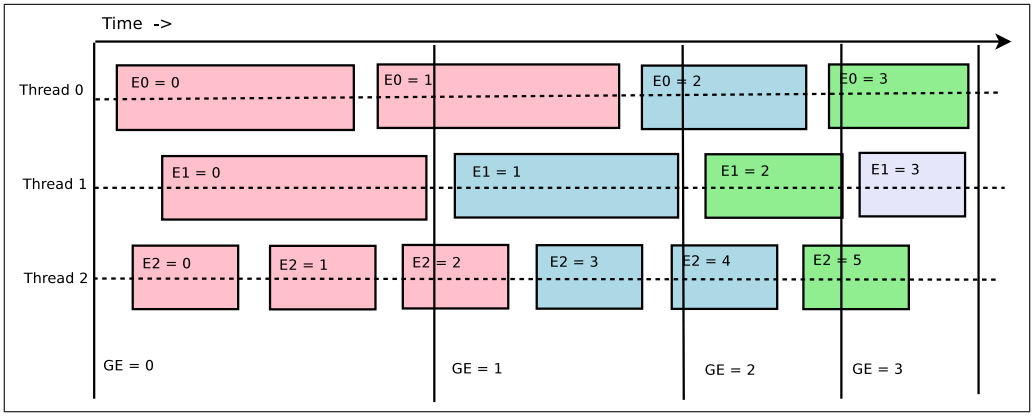

Figure 5: Detecting quiescent states through global epochs

Implementation details In order to maintain epoch numbers we introduce a *_thread_epoch* variable in the RouterThread class. The Master maintains the global epoch number. When control is returned to the driver loop after tasks are run, we increment the *_thread_epoch* variable and attempt to reclaim freed data. As in the Timer approach described earlier, we maintain two lists: a *reclaim_now list* and a *reclaim_later* list. An updater always adds any stale data to the *relcaim_later* list. During a reclaimation, we free the data structures in the *reclaim_now* list and swap the two lists. This ensures that we follow the freeing mechanism described in the above paragraph. Effectively, we can reclaim freed-data if every thread has changed its epoch variable since the last successful reclamation or if thread is blocked.

6.6 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 Read-Write lock.

Feature	Value
CPU	Intel(R) Core i3-370M (2.4 GHz)
kernel	Linux 2.6.32-25-generic
cache-line size	64 bytes
L1 cache size (per-cpu)	32 KB
L2 cache size	512 KB
L3 cache size	3072 KB

Table 2: Machine Characteristics

No. of Reader Threads	No Locks (s)	Reader-Writer Lock (s)	RCU (s)
1	0.565	0.66125	0.637
2	0.6125	4.3755	0.685
3	0.725	6.207	0.76
4	0.8925	11.585	0.7925

Table 3: Performance comparison over a pure reader workload

7 Performance Evaluation

7.1 Experimental Setup

We analyzed the performance of our solutions on a commodity Dell Studio 15 machine. The characteristics of this machine is shown in Table 7.1.
In our tests, we used a shell scripts which runs each Click script for a specified number of iterations .The CPU time is the average system time over 20 iterations.

7.2 Micro-benchmarks

The micro-benchmarks have tests which repeatedly access a small set of addresses in a very short time span. This was done by creating an element in Click which has a task of repeatedly reading a route.

Pure Reader Workload We look at how the coarse-grained reader-writer lock and RCU perform against the unmodified version which runs only readers. The unmodified version which can run only readers is used as a base-line reference against the solutions we have experimented with. As we can see from Table 7.2 which is represented in Figure 6, the reader-writer lock is upto 13 times slower than a version without locks. The reader-writer lock does not scale with an increase in the number of threads. This can be seen clearly in Figure 6. The Click configuration file used for this purpose is shown in Listing 15.
We also see that the RCU version scales with the number of threads. The RCU version is also within 15% of the performance of the version without any locks.

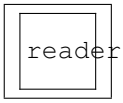

Figure 6: Performance comparison over a pure reader workload

```
1 // 4 readers using the RCU (fine grained locking version)
2 // of RadixIPLookup.
3 Idle
4  -> r :: RadixIPLookup106(
5                 1.1.1.0/32  8.1.1.1 0,
6                 0.0.0.0/0   8.1.1.1 0,
7                 )
8  -> Idle;
9
10 reader :: ReadRadixIPLookup106(r);
11 reader1 :: ReadRadixIPLookup106(r);
12 reader2 :: ReadRadixIPLookup106(r);
13 reader3 :: ReadRadixIPLookup106(r);
14 StaticThreadSched(
15         reader 0,
16         reader1 1,
17         reader2 2,
18         reader3 3
19 );
20
21 DriverManager(stop);
```

Listing 15: A Click configuration file for 4 readers

Workload	Reader-Writer Lock (s)	RCU (s)	Percentage Improvement (%)
1 R, 1 W	1.3425	0.78	41.8994413407821
2 R, 1 W	5.1025	0.945	81.4796668299853
3 R, 1W	8.535	1.975	76.8599882835384

Table 4: Performance comparison over a pure reader workload

Read-Intensive workloads We now compare the performance of RCU over the reader-writer lock in the presence of an updater. We refer the reader to Table 7.2 for the numbers. From Figure 7 see that the reader-writer lock does not scale with an increase in the number of readers. For read intensive work load consisting of 3 readers and 1 writer RCU performs 76% better than a reader-writer lock. The third column compares RCU considering the reader-writer lock as a baseline. The percentage improvement column computes $\frac{RWLock-RCU}{RWLock} \times 100$. Each of the readers execute a task consisting of reading a particular route 100000 times. Each task is executed 128 times.

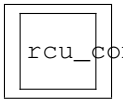

Figure 7: Performance comparison with one writer and increasing number of readers.

Write Intensive workloads We now compare the performance of RCU over the reader-writer lock with a write intensive workload. Our goal was to improve performance over read intensive workloads, however we include this section for completeness and correctness. From Table 7.2 and Figure 8, we can see that time taken by RCU begins to climb and exceeds the time taken by the reader-writer lock. This could be due to the additional overhead of reclaiming stale data in quiescent states. When there are more writers, there is more stale data.
The percentage improvement column computes $\frac{RWLock-RCU}{RWLock} \times 100$.

Workload	Reader-Writer Lock (s)	RCU (s)	Percentage Improvement (%)
1 R, 1 W	1.3425	0.78	41.8994413407821
2 R, 1 W	5.1025	0.945	81.4796668299853
3 R, 1W	8.535	1.975	76.8599882835384

Table 5: Performance comparison over a pure reader workload

Figure 8: Performance comparison with one reader and increasing number of writers.

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.

7.3 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 but have a lesser fraction of writes.

We used a realistic routing table derived from the routeviews.org database. 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 are 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. The machine used for the benchmarks has the configuration shown in Table 6.

Machine configuration will be moved up once microbenchmarks are run on the same machine

7.3.1 Pure Reader Workload

We first look at a workload consisting of only readers. The results are shown in Figure 9 and Table 7.

Workload	No Locks (s)	RW Lock (s)	RCU (s)
1 reader(s), 0 writer(s)	0.903	2.157	1.473
2 reader(s), 0 writer(s)	0.930	3.727	1.470
3 reader(s), 0 writer(s)	0.947	7.530	1.500
4 reader(s), 0 writer(s)	1.027	11.797	2.043
5 reader(s), 0 writer(s)	1.223	14.287	1.690
6 reader(s), 0 writer(s)	1.350	18.233	1.730
7 reader(s), 0 writer(s)	1.410	22.283	1.750
8 reader(s), 0 writer(s)	1.503	45.257	2.327

Table 7: Performance comparison over a pure reader workload.

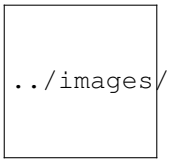

Figure 9: Performance of increasing number of readers using the 167k routing table.

From Figure 9, we see that RCU scales much better than the reader-writer lock version. RCU is always within 1.6 times the vanilla version. Time taken by the reader-writer lock increases linearly and is up to 31 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. Readers in the reader-writer lock version 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. Thus we see a linear increase in time as the number of readers increase. **Justify with profiling measurements.

7.3.2 Read Intensive Workload

We also conducted a benchmark which consisted of one updater and an increasing number of readers. The results are shown in Figure 10 and Table 8.

We see from Figure 10 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. Thus we see that RCU outperforms the reader-writer lock for a read intensive workload with lesser updates.

**Profile reader-writer lock and RCU. Then explain the linear increase in time taken by the reader-writer lock.

Workload	RW Lock (s)	RCU (s)
1 reader(s), 1 writer(s)	2.280	1.550
2 reader(s), 1 writer(s)	3.053	1.563
3 reader(s), 1 writer(s)	7.520	1.617
4 reader(s), 1 writer(s)	10.490	1.697
5 reader(s), 1 writer(s)	14.307	1.783
6 reader(s), 1 writer(s)	18.200	1.857
7 reader(s), 1 writer(s)	22.553	2.373

Table 8: Performance comparison of readers with one writer.

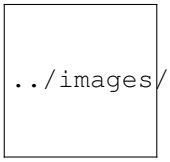

Figure 10: Performance of increasing number of readers and one writer with the 167k routing table.

8 Conclusion

Lookup performance in a software router such as Click is crucially important. It becomes necessary to leverage the power of multicore processors which are available in almost all commodity hardware today. A fast and safe multicore solution is desirable and this document has outlined an efficient way to achieve it.

We have built an RCU framework for userlevel Click invlolving a mechanism to detect quiescent states and an API which can be used by any userlevel element. We have verified through our benchmarks that this mechanism does indeed have a very low reader side overhead (within 10% of the original version). The RCU approach is also up to 70% better than a reader-writer lock on read intensive workloads.

The primary challenges in implementing RCU for Click were to identify quiescent states in a way which can be applied to all elements. We accomplished this through the Click scheduling loop.

Our analysis shows that RCU outperforms locking by a huge margin for read intensive workloads while being comparable to locking for an update intensive workload.

9 Future Work

1. Implement RCU for RadixIPLookup if Radix nodes are deleted when a route is removed.

2. Implement RCU for the Radix Tree: Currently, RCU has been only used on the vector. A coarse grained lock is acquired around updates to the Radix Tree.

10 Acknowledgements

I am tremendously grateful to my advisor, Professor Eddie Kohler for his continued guidance, support and ideas. I am grateful to Rohit Krishna Kumar for insightful discussions, code and comments.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.