RCU vs. Locking Performance on Different CPUs

Paul E. McKenney
IBM Corporation

Abstract

RCU has provided performance benefits in a number of areas in the Linux 2.6 kernel [MAK$^+$01, LSS02, MSA$^+$02, ACMS03, McK03, MSS04], as well as in several other operating system kernels [HOS89, Jac93, MS98, GKAS99]. Its use has also been proposed in conjunction with a number of specific algorithms [KL80, ML84, Pug90]. This experience has generated a number of useful rules of thumb, analytic comparisons of RCU to other locking techniques [MS98, McK99], and system-wide comparisons of specific RCU patches to the Linux kernel. However, there have not been any measured comparisons of RCU to other locking techniques under a variety of conditions running on different CPUs.

This paper fills that gap, comparing RCU to other locking techniques on a number of CPUs using a hash-lookup micro-benchmark. The read intensity, number of CPUs, and number of searches/updates per grace period are varied to gain better insight into which tool is right for a particular job.

1 Introduction

A useful comparison of RCU and other locking primitives requires the following:

1. Repeatability.
2. Relatively short duration to permit testing many combinations.
3. Code sequences that occur in real life.
4. Conservative treatment of the "new kid on the block," in this case, RCU.

Since the Linux kernel contains a very large number of hash tables, hash-table search is a natural choice for a mini-benchmark. To promote repeatability, deletions and insertions are performed in pairs, so that the number of elements in each hash chain remains constant over the long term–over the short term, of course, one CPU might search the hash table between the time after some other CPU has deleted an element, but before it has re-inserted it.

In order to err on the conservative side, this benchmark is compiled without optimization, which de-emphasizes the cache-thrashing overhead of the locking primitives. Comparison with measured memory-latency and pipeline-flush overheads showed that these costs dominated, in any case. This benchmark also disregards the possibility of batching RCU updates among multiple hash tables.

The locking methods compared are as follows:

1. Global spinlock (global).
2. Global rwlock (globalrw).
3. Per-bucket spinlock (bkt).
4. Per-bucket rwlock, but modified to avoid starvation (bktrw).
5. brlock.
6. Per-bucket spinlock combined with a per-element reference count (refcnt). Note that this mechanism helps with deadlock avoidance.
7. RCU, which entails lock-free search with per-bucket spinlock and RCU guarding updates. Note that this mechanism also helps with deadlock avoidance for read-side accesses.
8. Ideal scaling, taking the performance of searches and updates running without any sort of synchronization, and multiplying by the number CPUs. Once someone achieves ideal scaling on a given workload, that workload is thereafter designated "embarrassingly parallel".

This benchmark is run with from one to four CPUs, varying operation mixtures from read-only to update-only. Note that the single-CPU data still uses locks–see the "ideal" lines for lock-free performance. From this data, we will identify the areas of optimality for each locking primitive. Although these results are quite useful as rules of thumb, they are certainly no substitute for actually running real-world benchmarks before and after making changes!

Finally, in keeping with Linus Torvalds's request at Ottawa last summer that people place a greater focus on user-level code, these benchmarks run at user level. As a result, some of the names of some of the atomic primitives differ slightly from their kernel equivalents in the code fragments that follow.

Section 2 provides background on RCU. Section 3 describes the hash-table mini-benchmark, including the code for the various locking mechanisms tested. Section 4 displays the measured results, and Section 5 presents summary and conclusions.

2 Background

This section gives a brief overview of RCU; more details are available elsewhere [MS98, MAK$^+$01, MSA$^+$02].

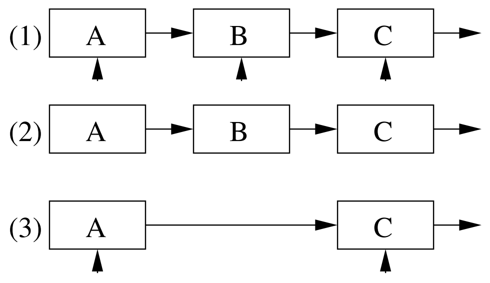

Figure 1: Lock Protecting Deletion and Search

On SMP systems, any searching of or deletion from a linked list must be protected, for example, by a lock. When element B is deleted from the list shown in Figure 1, searching code is guaranteed to see this list in either the initial state (1) or the final state (3). In state (2), when element B is being deleted, the reader-writer lock guarantees that no readers (indicated by the absence of a triangle on element B) will be accessing the list.

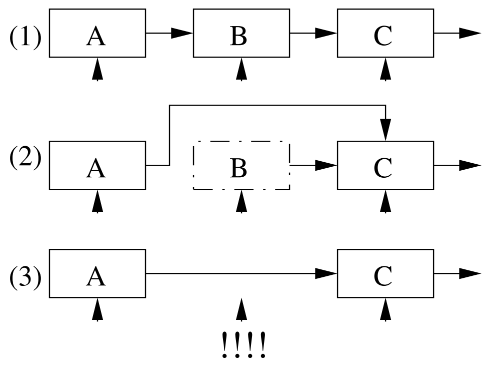

Figure 2: Race Between Deletion and Search

However, many lists are searched much more often than they are modified. For example, an IP routing table would normally change at most once per few minutes, but might be searched many thousands of times per second. This could result in well over a million accesses per update, making lock-acquisition overhead burdensome to searches.

Unfortunately, omitting locking when searching means that the update no longer appears to be atomic. Instead, the update takes the multiple steps shown in Figure 2. A search might be referencing element B just as it was freed up, resulting in crashes, or worse, as indicated by the reader referencing nothingness in step (3).

One solution to this problem is to delay freeing up element B until all searches have given up their references to it, as shown in Figure 3. RCU indirectly determines when all references have been given up. To see how this works, recall that there are normally restrictions on what

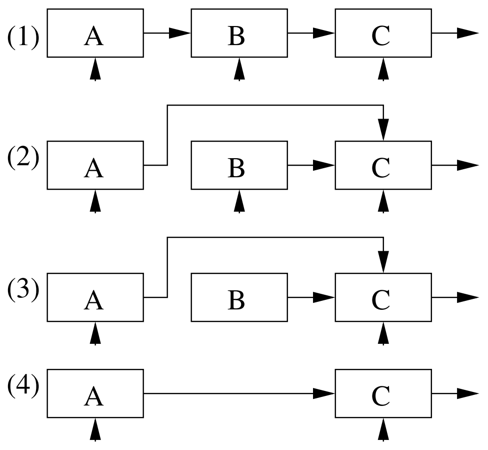

Figure 3: RCU Protecting Deletion and Search

operations may be performed while holding a lock. For example, in the Linux kernel, it is forbidden to do a context switch while holding any spinlock. RCU mandates these same restrictions: even though the RCU-protected search need not acquire any locks, it is forbidden from performing any operation that would be forbidden if it were in fact holding a lock.

Therefore, any CPU that is seen performing a context switch after the linked-list deletion shown in step (2) of Figure 3 cannot possibly hold a reference to element B. As soon as all CPUs have performed a context switch, there can no longer be any readers, as shown in step (3). Element B may then be safely freed, as shown in step (4).

A simple, though inefficient, RCU-based deletion algorithm could perform the following steps in a non-preemptive Linux kernel (preemptive kernels can be handled as well [MSA$^+$02]):

1. Unlink element B from the list, but do *not* free it. The state of the list will be that shown in step (2) of Figure 3.
2. Run on each CPU in turn. At this point, each CPU has performed one context switch after element B has been unlinked. Thus, there cannot be any more references to element B.
3. Free up element B.

Much more efficient implementations have been described elsewhere [MS98, MSA$^+$02].

3 Hash-Table Mini-Benchmark

The hash table is a simple array of buckets, with linked-list chains. Separate tasks perform searches and updates in parallel, and a parent task terminates the test after the prescribed duration. Each experiment runs the test five times, and prints the average, maximum, and minimum times per-operation times from the five runs, along

```
struct el *
_search_bucket(struct list_head *p, long key)
{
  struct list_head *lep;
  struct el *q;

  list_for_each(lep, p) {
    q = list_entry(lep, struct el, list);
    if (q->key == key) {
      return (q);
    }
  }
  return (NULL);
}

struct el *
_search(long key)
{
  struct list_head *p;

  p = KEY2CHAIN(key);
  return (_search_bucket(p, key));
}
```

Figure 4: Search Functions

with the standard deviation. RCU-based tests print the same statistics for the maximum number of hash-table elements that were ever waiting for a grace period to complete.

The following section describes the common code used in the test, and the sections after that describe code specific to each of the locking mechanisms.

3.1 Common Code

Figure 4 shows the code for the generic search functions, which is quite straightforward. The `search()` function is used by mechanisms such as RCU and brlock that do not use a per-hash-chain lock, while the `_search_bucket()` is used by mechanisms that use a per-hash-chain lock. The `KEY2CHAIN()` macro provides a simple modulus hash function.

Figure 5 shows the test loop. The keys of the elements in the hash table are consecutive integers, and each task increments its key by a different odd prime on each search or update, in order to avoid "convoy" effects. The `INSERT()`, `SEARCHSHD()`, and `SEARCHX()` macros map directly to their lower-case equivalents for the locking primitives compared in this paper. This extra level of C-preprocessor indirection will allow future comparison of wait-free synchronization algorithms. The `nsearch` variable on line 3 contains the expected number of searches in the search/update mix; similarly, the `nmodify` variable indicates the expected number of updates in the mix, so that a mixture of 90% searches might have `nsearch=9` and `nmodify=1`. The loop from lines 4-13 performs a randomly selected number of searches based on the desired mix. The `SEARCHSHD()` function on line 5 is defined to return with any required locks or reference counts held, so line 7 invokes `releaseshd()` to do any necessary releasing. Lines 9-12 increment the search key with wrap, and line 14 counts the number of operations in a local variable.

Similarly, the loop from lines 16-38 of Figure 5 performs a randomly selected number of updates based on the desired mix. If the `SEARCHX()` function on line 18 returns non-NULL, it is defined to return holding whatever locks are required to delete the element, and the `DELETE()` function on line 21 is defined to drop those locks before return, and free the deleted element as well. On the other hand, if `SEARCHX()` returns NULL, it returns with no locks held. Line 22 allocates a new element to replace the one deleted, and line 24 counts an error if out of memory. Line 26 does some debug checks if RCU is in use, and is a no-op otherwise, again favoring normal locking. Lines 27-30 initialize the new element and insert it. The `INSERT()` function on line 29 is defined to acquire any needed locks and to release them before returning. Lines 34-37 advance to the next key value, and line 39 counts the operations in a local variable. Line 40 forces a quiescent state for RCU, and is a no-op otherwise.

As noted earlier, this test loop is run five times for each configuration, and the results are averaged. The maximum, minimum, and standard deviation are also computed in order to check repeatability and detect interference.

3.2 Starvation-Free rwlock

In order for the tests to run reliably, a starvation-free rwlock is required. Similar implementations have been reinvented many times over the decades, the first that I am aware of being Courtois et al. [CHP71]. Please note that I am *not* advocating that the Linux kernel also adopt a starvation-free rwlock, since freedom from starvation incurs additional overhead in low-contention situations. Use the right tool for the job!

The rwlock is implemented as an atomically manipulated integer that takes on values as follows:

1. `RW_LOCK_BIAS`: lock is idle. This is the initial value.
2. `RW_LOCK_BIAS`$-n$, where n is less than `RW_LOCK_BIAS`/2: n readers hold the lock.
3. 0: one writer holds the lock.
4. $-n$: one writer is waiting for n readers to release the lock.

Figure 6 shows the acquisition procedure. The outer loop from lines 6-16 spins until we have read-acquired the lock. The inner loop from lines 9-11 spins until any current write-holder has released the lock, and also takes a snapshot of the lock value at this point. Lines 15-16 then attempt to read-acquire the lock. Finally, line 17 issues whatever memory barrier instructions are required

```
 1 key = 0;
 2 while (atomic_read4(&shdctl->state) == STATE_GO) {
 3   n = random() % (2 * nsearch + 1);
 4   for (i = 0; i < n; i++) {
 5     q = SEARCHSHD(key);
 6     if (q != NULL) {
 7       releaseshd(q);
 8     }
 9     key += incr;
10     if (key >= nelements) {
11       key -= nelements;
12     }
13   }
14   mycount += n;
15
16   n = random() % (2 * nmodify + 1);
17   for (i = 0; i < n; i++) {
18     q = SEARCHX(key);
19     if (q != NULL) {
20       key = q->key;
21       DELETE(q);
22       q = kmalloc(sizeof(*q), 0);
23       if (q == NULL) {
24         shdctl->pt[id].err_count++;
25       } else {
26         RCU_VALIDATE_ALLOC(q, key);
27         init_el(q);
28         q->key = key;
29         if (!INSERT(q)) {
30           kfree(q);
31         }
32       }
33     }
34     key += incr;
35     if (key >= nelements) {
36       key -= nelements;
37     }
38   }
39   mycount += n;
40   RCU_QS();
41 }
```

Figure 5: Test Loop

```
 1 static inline void
 2 read_lock(atomic_t *rw)
 3 {
 4   long oldval;
 5
 6   do {
 7     /* Wait for writer to get done.  */
 8
 9     do {
10       oldval = (long)atomic_read4(rw);
11     } while (oldval <= RW_LOCK_BIAS / 2);
12
13     /* Attempt to read-acquire the lock. */
14
15   } while (atomic_cmpxchg4(rw,oldval,oldval-1) !=
16            oldval);
17   spin_lock_barrier();
18 }
```

Figure 6: Read-Acquiring an rwlock

```
 1 static inline void
 2 read_unlock(atomic_t *rw)
 3 {
 4   spin_unlock_barrier();
 5   (void)atomic_xadd4(rw, 1);
 6 }
```

Figure 7: Read-Releasing an rwlock

to prevent the critical section from “bleeding out” into the preceding code.

Figure 7 shows how an rwlock is released. Line 4 executes any memory barriers required to prevent the critical section from bleeding out into the following code, and line 6 atomically increments the lock variable, indicating that one fewer reader is holding the lock.

Figure 8 shows the procedure to write-acquire the rwlock. The outer loop spanning lines 7-19 spins repeatedly attempting to write-acquire the lock, contending with any other concurrent writers. The inner loop spanning lines 11-14 repeatedly takes a snapshot of the lock variable and subtracts the bias, spinning until the result indicates that any preceding writer has released the lock. Once any preceding writer has released the lock, lines 18-19 attempt to atomically update the lock variable to indicate that this CPU is write-holding the lock. Then the loop from lines 23-25 spins waiting for any read-holding CPUs to release the lock. Finally, line 26 executes whatever memory-barrier instructions are required to prevent the critical section from bleeding out into the preceding code.

Figure 9 shows the code to release an rwlock. This is very similar to the read-release code. Line 4 executes whatever memory-barrier instructions are needed to prevent the critical section from bleeding out into the following code, and line 5 atomically adds the bias back into the lock variable.

Note that once a writer releases the lock, either a reader or a writer might acquire it next, so that a series of writers cannot deterministically starve a reader. Once

```
static inline void
write_lock(atomic_t *rw)
{
  long newval;
  long oldval;

  do {

    /* Wait for previous writer to finish. */

    do {
      oldval = (long)atomic_read4(rw);
      newval = oldval - RW_LOCK_BIAS;
    } while (newval < -RW_LOCK_BIAS / 2);

    /* Attempt to write-acquire. */

  } while (atomic_cmpxchg4(rw,oldval,newval) !=
           oldval);

  /* Wait for any readers to finish. */

  while (((long)atomic_read4(rw)) < 0) {
    continue;
  }
  spin_lock_barrier();
}
```

Figure 8: Write-Acquiring an rwlock

```
static inline void
write_unlock(atomic_t *rw)
{
  spin_unlock_barrier();
  (void)atomic_xadd4(rw, RW_LOCK_BIAS);
}
```

Figure 9: Write-Releasing an rwlock

```
struct el *
searchshd(long key)
{
  struct el *q;

  spin_lock(&(KEY2BUCKET(key)->bktlock));
  q = _search(key);
  if (q != NULL) {
    return (q);
  }
  spin_unlock(&(KEY2BUCKET(key)->bktlock));
  return (NULL);
}
```

Figure 10: Per-Bucket Search

```
1 void
2 delete(struct el *q)
3 {
4   struct list_head *p = &(q->list);
5
6   list_del(p);
7   releasex(q);
8   kfree(q);
9 }
```

Figure 11: Per-Bucket Delete

a writer makes its presence known, it will eventually obtain the lock, so that readers cannot starve a writer.

3.3 Per-Bucket Spinlock

Figure 10 shows the per-bucket-spinlock search function. It simply computes the hash, acquires the spinlock for the corresponding hash bucket, searches the hash chain, and returns with the lock held for a successful search, or drops the lock (and returns NULL) for an unsuccessful search. The `searchx()` function is identical.

Figure 11 shows the `delete()` function, which is passed an element returned by a successful call to `searchx()`, and is thus invoked with the bucket lock held. Line 6 removes the element from the hash chain, line 7 releases the bucket lock, and line 8 frees up the newly removed element.

Figure 12 shows the `insert()` function. Line 6 acquires the bucket lock. Lines 7-11 verify that the element's key is not already in the list, dropping the lock and returning failure if it is. If the key is not already in the list, line 12 adds the new element to the list, line 13 releases the lock, and line 14 returns success.

3.4 Per-Bucket rwlock

The per-bucket rwlock's `searchshd()` function is the same as that shown in Section 3.3, but with the spinlock operations replaced with read-side rwlock operations. Similarly, the per-bucket rwlock's `searchx()`, `insert()`, and `delete()` functions are the same as those shown in Section 3.3, with the spinlock operations replaced with write-side rwlock operations.

```
 1 int
 2 insert(struct el *q)
 3 {
 4   struct el *p;
 5
 6   spin_lock(&(KEY2BUCKET(q->key)->bktlock));
 7   p = _search(q->key);
 8   if (p != NULL) {
 9     spin_unlock(&(KEY2BUCKET(q->key)->bktlock));
10     return (0);
11   }
12   list_add(&(q->list), KEY2CHAIN(q->key));
13   spin_unlock(&(KEY2BUCKET(q->key)->bktlock));
14   return (1);
15 }
```

Figure 12: Per-Bucket Insert

```
 1 struct el *
 2 searchshd(long key)
 3 {
 4   struct el *q;
 5
 6   spin_lock(&(KEY2BUCKET(key)->bktlock));
 7   q = _search(key);
 8   if (q != NULL) {
 9     (void)atomic_xadd4(&q->refcnt, 1);
10     spin_unlock(&(KEY2BUCKET(key)->bktlock));
11     return (q);
12   }
13   spin_unlock(&(KEY2BUCKET(key)->bktlock));
14   return (NULL);
15 }
```

Figure 14: Reference-Count Shared Search

```
1 void
2 releaseshd(struct el *q)
3 {
4   if (atomic_xadd4(&q->refcnt, -1) == 1) {
5     kfree(q);
6   }
7 }
```

Figure 13: Reference-Count Release

```
1 void
2 delete(struct el *q)
3 {
4   struct list_head *p = &(q->list);
5
6   list_del(p);
7   releasex(q);  /* for search. */
8   releaseshd(q);  /* for list. */
9 }
```

Figure 15: Reference-Count Delete

3.5 Per-Bucket Spinlock and Per-Element Refcnt

Figure 13 shows how a reference count is released by `releaseshd()`. Line 4 atomically decrements the count and checks to see if the old value was 1, in which case this is the last reference, and the item may now be freed by line 5. Note that the linked list itself holds a reference, so that the element cannot possibly be freed until after a successful `delete()`.

The `releasex()` function simply releases the per-bucket lock.

Figure 14 shows the `searchshd()` function, which returns with the reference count held upon successful search. Line 6 acquires the per-bucket spinlock, and line 7 searches for the specified key. If the search is successful, line 9 increments the reference count, line 10 releases the per-bucket spinlock, and line 11 returns a pointer to the element. Otherwise, if the search is unsuccessful, line 13 releases the per-bucket spinlock and line 14 returns NULL.

Note that both the search and the reference-count increment are done under the per-bucket lock. Since deletion is also done under the lock, it is not possible to gain a reference to an element that has already been removed from its list.

The `searchx()` exclusive-lock search function is identical in function to the per-bucket-spinlock version shown in Figure 10. Upon successful search, it returns with the per-bucket lock held.

Figure 15 shows the reference-count `delete()` function. This function must be passed an element that was returned from `searchx()`, so that the per-bucket lock is held on entry to `delete()`. In addition, since the linked list itself holds a reference to the element, we know that the reference count value must be at least one, even if there are no references obtained from `searchshd()` currently in force. Therefore, line 7 simply removes the element from the list, line 8 drops the per-bucket lock, and line 9 releases the linked list's reference to the element. If there are no outstanding `searchshd()` references, the `releaseshd()` invocation on line 9 will also free up the element, otherwise, the last `searchshd()` reference to be dropped will free up the element.

Figure 16 shows the reference-count `insert()` function. This function is identical to the per-bucket-spinlock version shown in Figure 12, with the addition of the initialization of the reference count to 1 (for the linked list, remember?) on line 13.

```
 1 int
 2 insert(struct el *q)
 3 {
 4   struct el *p;
 5
 6   spin_lock(&(KEY2BUCKET(q->key)->bktlock));
 7   p = _search(q->key);
 8   if (p != NULL) {
 9     spin_unlock(&(KEY2BUCKET(q->key)->bktlock));
10     return (0);
11   }
12   atomic_set4(&q->refcnt, 1);
13   list_add(&(q->list), KEY2CHAIN(q->key));
14   spin_unlock(&(KEY2BUCKET(q->key)->bktlock));
15   return (1);
16 }
```

Figure 16: Reference-Count Insert

```
 1 struct el *
 2 searchshd(long key)
 3 {
 4   struct el *q;
 5
 6   br_read_lock(bbrlock);
 7   q = _search(key);
 8   if (q != NULL) {
 9     return (q);
10   }
11   br_read_unlock(bbrlock);
12   return (NULL);
13 }
```

Figure 17: brlock Shared Search

```
 1 void
 2 delete(struct el *q)
 3 {
 4   struct list_head *p = &(q->list);
 5
 6   list_del(p);
 7   releasex(q);
 8   kfree(q);
 9 }
```

Figure 18: brlock Delete

3.6 Big-Reader Lock

Figure 17 shows the brlock `searchshd()` function. Line 6 read-acquires the lock and line 7 performs the search. If this search is successful, line 9 returns a pointer to the element, while still holding the lock. Otherwise, line 11 drops the lock and line 12 returns NULL.

The only difference between the `searchx()` function and the `searchshd()` function is that `searchx()` write-acquires the brlock.

Figure 18 shows the brlock `delete()` function. Line 6 removes the specified from the list, line 7 write-releases the brlock, and line 8 frees the element.

Figure 19 shows the brlock `insert()` function, which is quite similar to the per-bucket-spinlock version shown in Figure 12, but with the per-bucket exclusive lock operations replaced with the corresponding write-side brlock operations.

```
 1 int
 2 insert(struct el *q)
 3 {
 4   struct el *p;
 5
 6   br_write_lock(bbrlock);
 7   p = _search(q->key);
 8   if (p != NULL) {
 9     br_write_unlock(bbrlock);
10     return (0);
11   }
12   list_add(&(q->list), KEY2CHAIN(q->key));
13   br_write_unlock(bbrlock);
14   return (1);
15 }
```

Figure 19: brlock Insert

```
 1 struct el *
 2 searchshd(long key)
 3 {
 4   struct el *q;
 5
 6   rcu_read_lock();
 7   q = _search(key);
 8   if (q != NULL) {
 9     return (q);
10   }
11   rcu_read_unlock();
12   return (NULL);
}
```

Figure 20: RCU Shared Search

```
 1 void
 2 delete(struct el *q, struct rcu_handle *rhp)
 3 {
 4   struct list_head *p = &(q->list);
 5
 6   list_del(p);
 7   RCU_SET_DELETED(q);
 8   releasex(q);
 9   call_rcu_kfree(rhp, q);
10 }
```

Figure 21: RCU Delete

3.7 RCU

Figure 20 shows the RCU `searchshd()` function, which does a simple search. Note that `rcu_read_lock()` and `rcu_read_unlock()` generate no code in this case. The RCU `searchx()` function is identical to its per-bucket-lock counterpart shown in Figure 10.

Figure 21 shows the RCU `delete()` function. Again, this function is called with the per-bucket lock held. Line 6 removes the specified element from the list, line 7 updates debug information, line 8 releases the per-bucket lock, and line 9 frees up the element at the end of the next grace period. The `call_rcu_kfree()` function also updates statistics that track the maximum number of elements waiting for a grace period to expire.

Figure 22 shows RCU's `insert()` function. Line 6 acquires the lock, and line 7 checks to see if the desired key is already present in the list. If it is, lines 9-10 release the lock and return failure. Otherwise, if the desired key is not already present, line 12 updates debug information, line 13 adds the element to the list with appropriate memory barriers, line 14 releases the lock, and line 15 returns success.

4 Measured Results

This section shows measured results from running the hash-table benchmark using the different locking methods on the different CPUs. The CPUs used are as follows:

1. Four-CPU 700MHz P-III system.
2. Four-CPU 1.4GHz IPF system.

```
int
insert(struct el *q)
{
  struct el *p;

  spin_lock(&(KEY2BUCKET(q->key)->bktlock));
  p = _search(q->key);
  if (p != NULL) {
    spin_unlock(&(KEY2BUCKET(q->key)->bktlock));
    return (0);
  }
  RCU_SET_INUSE(q);
  list_add_rcu(&(q->list), KEY2CHAIN(q->key));
  spin_unlock(&(KEY2BUCKET(q->key)->bktlock));
  return (1);
}
```

Figure 22: RCU Insert

3. Four-CPU 1.4GHz Opteron system.
4. Eight-CPU 1.45GHz POWER4+ system (but only four CPUs were used in these benchmarks).

Section 4.1 displays results for a read-only workload, while Section 4.2 shows how the breakevens vary as the update intensity of the workload varies.

4.1 Read-Only Workload

Figures 23, 24, 25, and 26 show performance results for each of the locking mechanisms. The y-axes are normalized in order to prevent inappropriate comparison of the different CPUs.

The left-hand graph of each figure shows the performance of a global spinlock and a global rwlock. In each case, these primitives give *negative* scaling. The cache misses incurred by rwlock overwhelm any read-side parallelism one might hope to gain. For rwlock to show decent scaling, the read-side critical sections must be quite lengthy.

The right-hand graph of each figure shows the performance of brlock, the per-bucket-locking mechanisms, and RCU. Note that RCU does not achieve ideal performance, even in this read-only benchmark. In fact, for some CPUs, the deviation from ideal is significant, in contrast to experience in the kernel, where the overhead of the checks is not measurable. This is due to the fact that this user-level benchmark does not have an existing `scheduler_tick()` function in which a low-cost test can be placed. User-level code therefore bears a greater cost for RCU than does does the kernel, so, once again, these comparisons are conservative.

4.2 Mixed Workload

The mixed-workload results are more interesting. Although RCU outperforms the other mechanisms in the read-only case, the extra overhead of detecting grace periods should slow it down for updates. RCU would therefore be expected to be optimal only up to a particular update fraction.

This section shows what this update fraction is for different values of λ, which is the expected number of searches and updates per grace period. The greater the value of λ, the greater the number of searches and updates to absorb the overhead of detecting a grace period. One would therefore expect RCU's breakeven update fraction to increase with increasing λ.

The two values chosen for λ are 10 and 100. These values are quite conservative given that realtime-response-time tests of the Linux 2.6.0 kernel under heavy load have observed more than 1,000 updates per grace period. The corresponding value of λ would be even greater, but the current kernel instrumentation does not measure the number of read-side RCU critical sections. However, even if the number of read-side critical sections is zero, the values of 10 and 100 used in this paper are quite conservative. This came as a surprise, as experience with RCU in other operating systems [MS98] has placed λ in the range from 0.1 to 10.

Figures 27 through 38 show the performance of the various locking mechanisms for 4-CPU systems of x86, IPF, Opteron, and PPC CPUs. In each figure, the graph on the left shows λ of 10, and the graph on the right shows λ of 100. Again, the y-axis is normalized to prevent inappropriate comparison between the different CPUs.

The crossover update fractions between RCU and per-bucket locking are summarized in Table 1. The crossover numbers in the table are approximate–note that the lines crossing are often nearly parallel, which means that small experimental errors translate to relatively large changes in the crossover number. The crossover update fractions are all rather large, with RCU remaining optimal when up to half of the accesses are updates for some CPUs. The x86 results show RCU improving with increasing numbers of CPUs, but Opteron, and particularly IPF/x86, show decreases. Future work for IPF includes running the test in the native instruction set to see if this behavior is an artifact of the x86 hardware emulation environment. In the case of Opteron, the effect is much smaller, but it would still be quite interesting to run this experiment on an 8-CPU Opteron system.

The "wavy" lines in a number of plots are due to run-to-run variation in measured performance.

RCU comes very close to the ideal performance in a number of cases, but not universally. This is why single-CPU benchmarks are a special challenge for RCU-based changes, and also why it is so important to subject RCU-based changes to single-CPU benchmarks.

It appears from this data that RCU should provide a performance benefit on most systems when the update fraction is less than 0.1 or 10%. One important exception to this rule of thumb is the case of a data structure that already uses RCU for some of its code paths. Since the

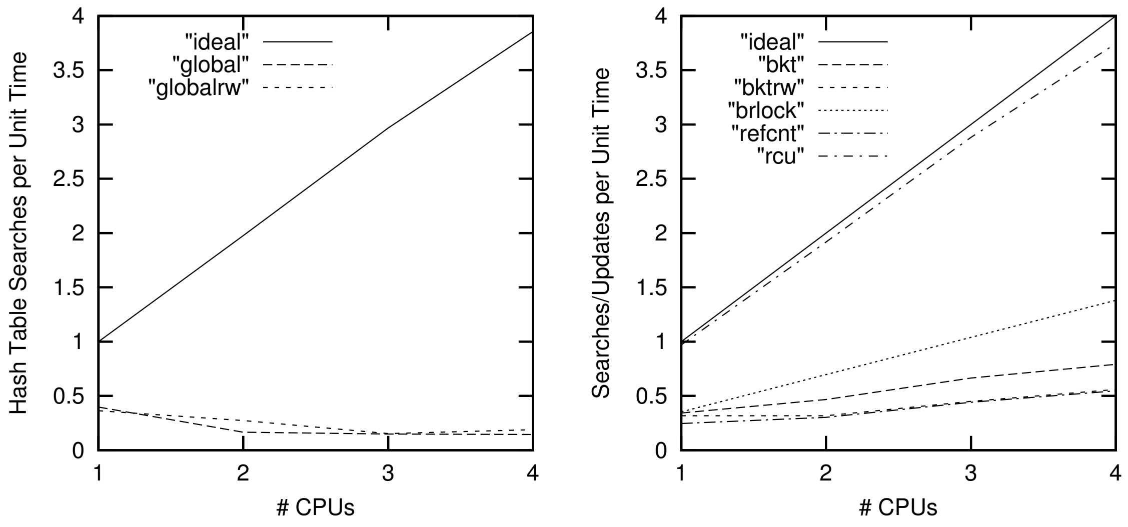

Figure 23: x86 Performance for Read-Only Workload

first use of RCU must bear the entire update cost, RCU may be used much more aggressively on additional code paths. As always, your mileage may vary–the data in this paper is in no way a substitute for careful performance evaluation under realistic conditions.

5 Summary

A hash-table mini-benchmark was used to evaluate the suitability of RCU for a range of update intensities on four different types of CPUs. As expected, RCU is best for read-only workloads. RCU remains best up to about a 10% update fraction on all systems under all tested conditions, and is best up to a 50% update fraction for 4-CPU x86, 1- and 2-CPU Opteron, and 2- and 4-CPU PPC for systems running with at least 100 searches/updates per grace period. Since machines running heavy workloads have been observed with more than 1,000 updates per grace period, these figures appear to be conservative.

Therefore, a conservative rule of thumb would be to use RCU when no more than about 10% of the accesses update the hash table. However, once a given data structure uses RCU on some code paths, RCU may be applied to its other code paths much more aggressively. Of course, your mileage may vary, so this paper is in no way a substitute for careful testing under realistic conditions. Update-intensive workloads generally do well with a per-bucket lock. In update-intensive situations where deadlock is a problem, it would be better to use reference counts with per-bucket locking. Note that the plots of performance data are qualitatively similar, as would be expected given that cacheline traffic is responsible for much of the overhead. Therefore, these rules of thumb are likely to be quite reliable.

Future work includes running on a greater variety of CPUs and on different models within the same CPU family, and testing the locking mechanisms on different data structures.

Acknowledgements

I am indebted to Jack Vogel, Chris McDermott, Dave Hansen, and Adam Litke, who provided time on the test systems. I owe thanks to Dan Frye, Jai Menon, and Juergen Deicke for their support of this work. As always, I look forward to continuing to learn about RCU and its uses, and am grateful to the many people who have tried it out. It is from them that I learn the most.

References

[ACMS03] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)*, June 2003.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with “readers” and “writers”. *Communications of the ACM*, 14(10):667–668, October 1971.

[GKAS99] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating

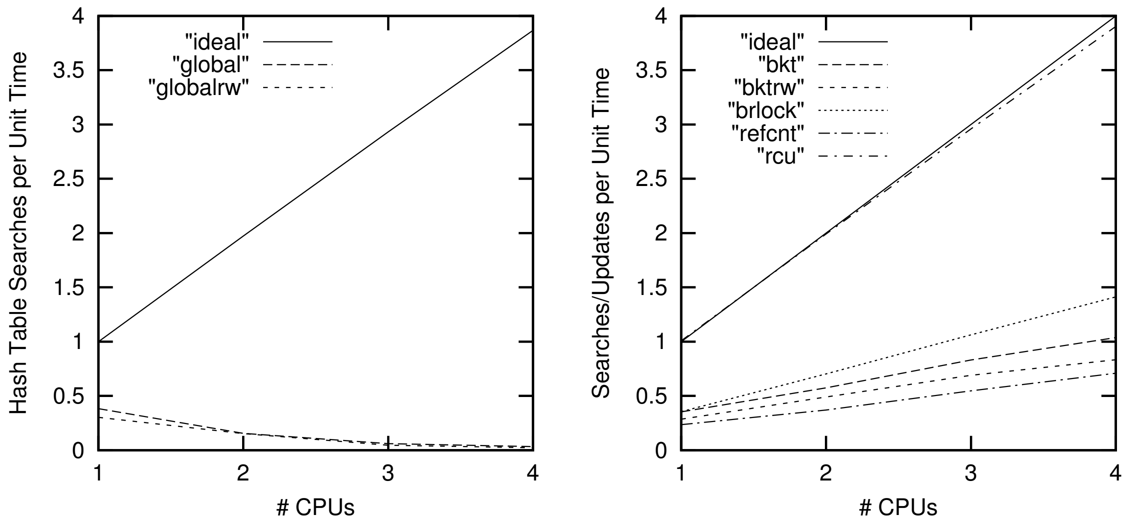

Figure 24: IPF/x86 Performance for Read-Only Workload

system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, New Orleans, LA, February 1999.

[HOS89] James P. Hennessey, Damian L. Osisek, and Joseph W. Seigh II. Passive serialization in a multitasking environment. Technical Report US Patent 4,809,168, US Patent and Trademark Office, Washington, DC, February 1989.

[Jac93] Van Jacobson. Avoid read-side locking via delayed free. Verbal discussion, September 1993.

[KL80] H. T. Kung and Q. Lehman. Concurrent maintenance of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, September 1980.

[LSS02] Hanna Linder, Dipankar Sarma, and Maneesh Soni. Scalability of the directory entry cache. In *Ottawa Linux Symposium*, pages 289–300, June 2002.

[MAK+01] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *Ottawa Linux Symposium*, July 2001.

[McK99] Paul E. McKenney. Practical performance estimation on shared-memory multiprocessors. In *Parallel and Distributed Computing and Systems*, pages 125–134, Boston, MA, November 1999.

[McK03] Paul E. McKenney. Using RCU in the Linux 2.5 kernel. *Linux Journal*, 1(114):18–26, October 2003.

[ML84] Udi Manber and Richard E. Ladner. Concurrency control in a dynamic search structure. *ACM Transactions on Database Systems*, 9(3):439–455, September 1984.

[MS98] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[MSA+02] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Ottawa Linux Symposium*, pages 338–367, June 2002.

[MSS04] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. Using rcu in the linux 2.6 directory-entry cache. *to appear in Linux Journal*, 1(118), January 2004.

[Pug90] William Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute of Advanced Computer Science Studies, Department of Computer Science, University of Maryland, College Park, Maryland, June 1990.

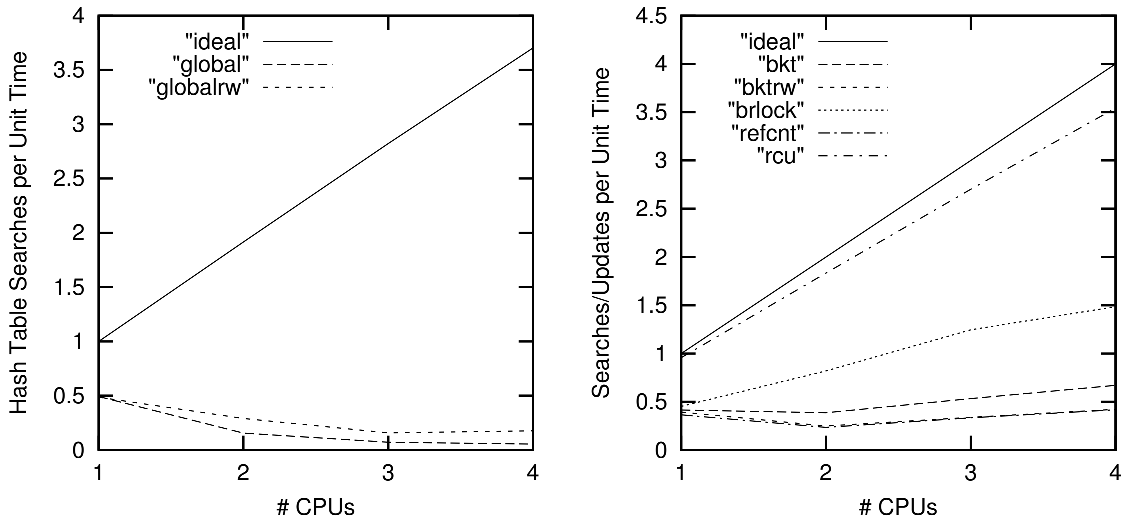

Figure 25: Opteron Performance for Read-Only Workload

Legal Statement

CPU	λ	# CPUs	Crossover	Figure
x86	10	1	0.2	27
		2	0.3	28
		4	0.3	29
	100	1	0.4	27
		2	0.4	28
		4	0.5	29
IPF/x86	10	1	0.3	30
		2	0.2	31
		4	0.1	32
	100	1	0.4	30
		2	0.4	31
		4	0.2	32
Opteron	10	1	0.3	33
		2	0.2	34
		4	0.2	35
	100	1	0.5	33
		2	0.5	34
		4	0.4	35
PPC	10	1	0.3	36
		2	0.5	37
		4	0.4	38
	100	1	0.4	36
		2	0.5	37
		4	0.5	38

Table 1: Summary of RCU Crossover Update Fractions

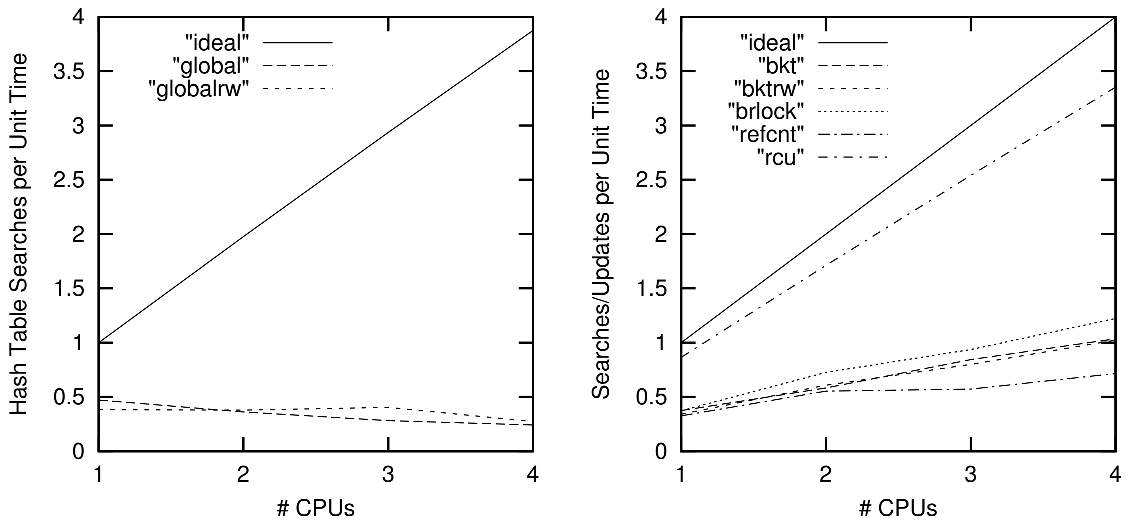

Figure 26: PPC Performance for Read-Only Workload

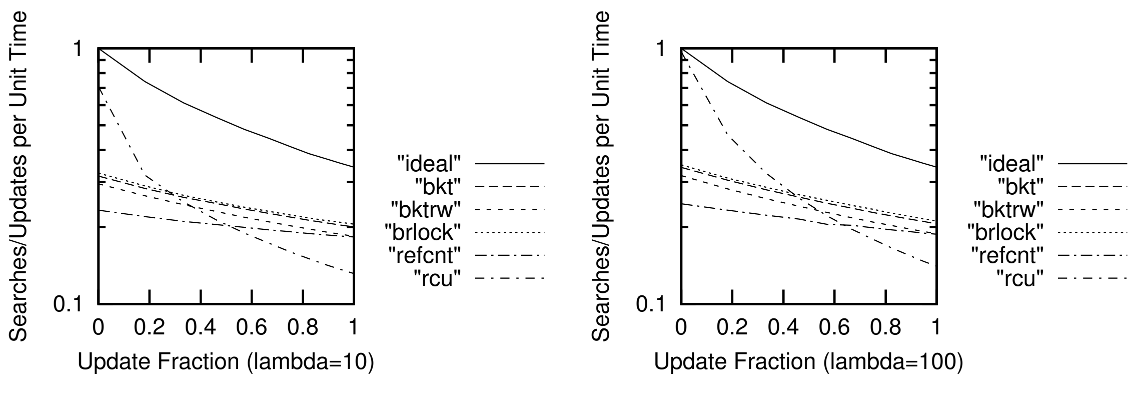

Figure 27: x86 1-CPU Performance for Mixed Workload

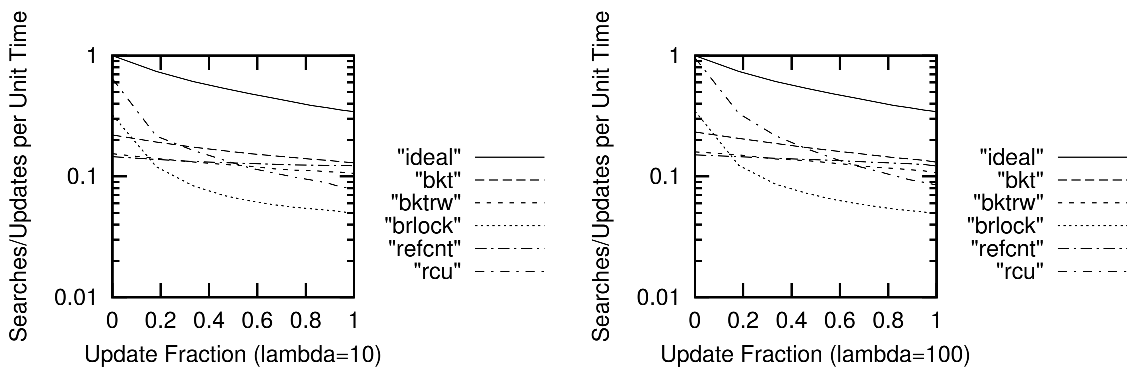

Figure 28: x86 2-CPU Performance for Mixed Workload

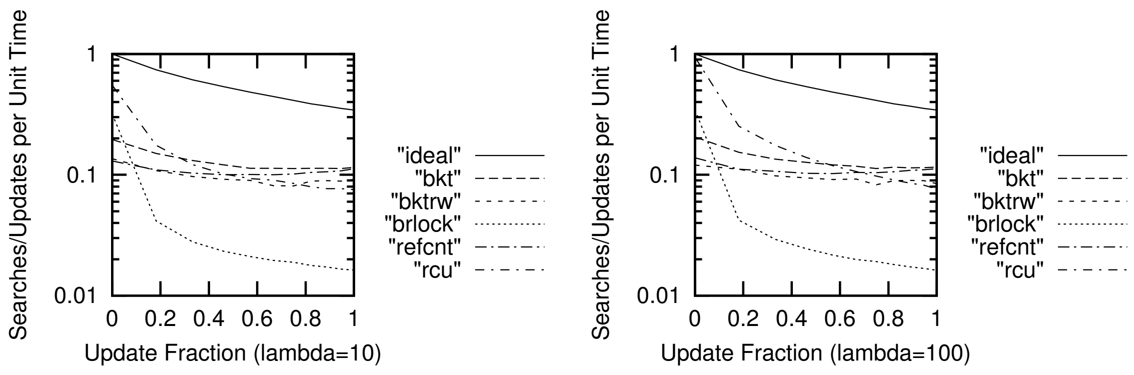

Figure 29: x86 4-CPU Performance for Mixed Workload

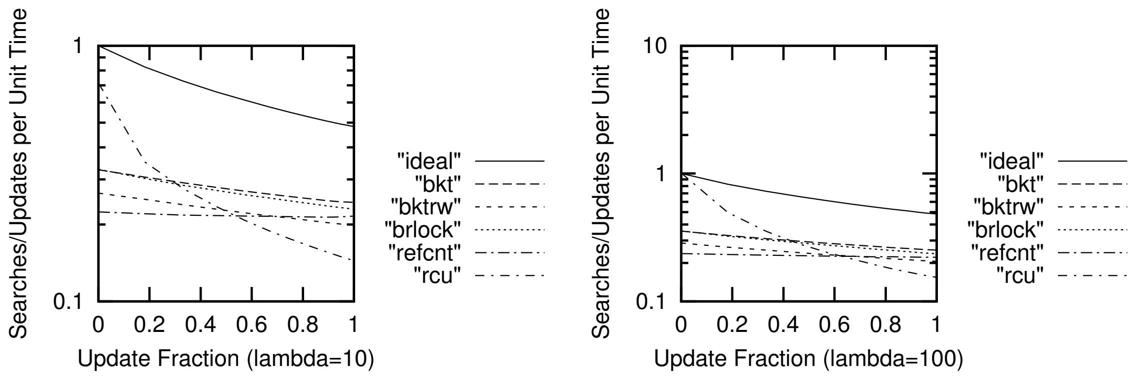

Figure 30: IPF/x86 1-CPU Performance for Mixed Workload

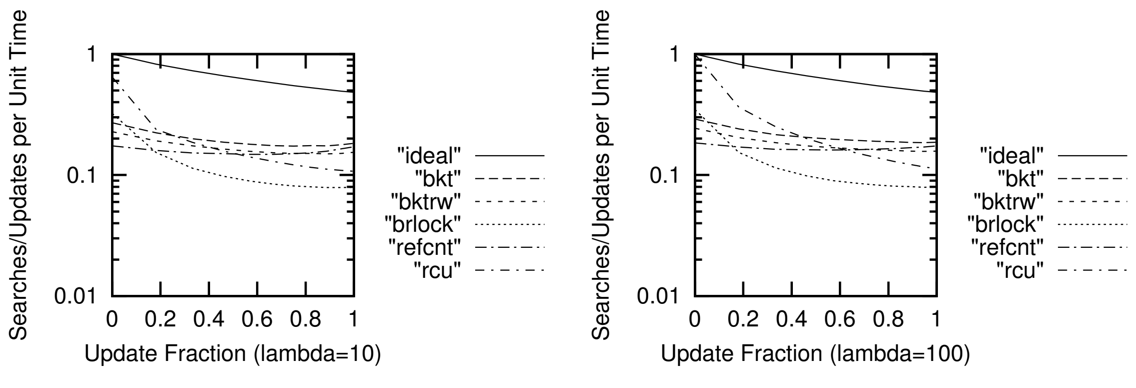

Figure 31: IPF/x86 2-CPU Performance for Mixed Workload

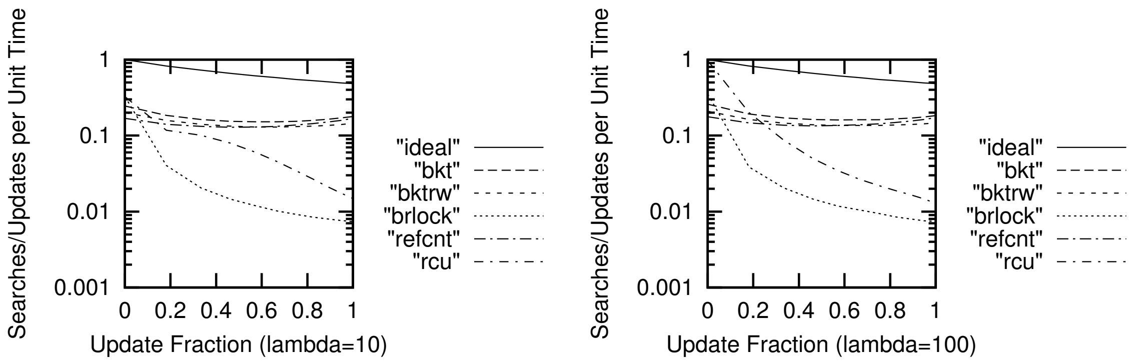

Figure 32: IPF/x86 4-CPU Performance for Mixed Workload

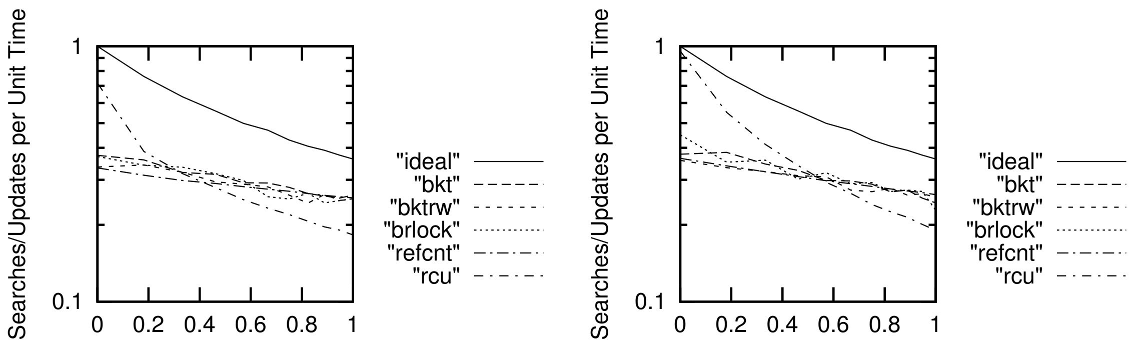

Figure 33: Opteron 1-CPU Performance for Mixed Workload

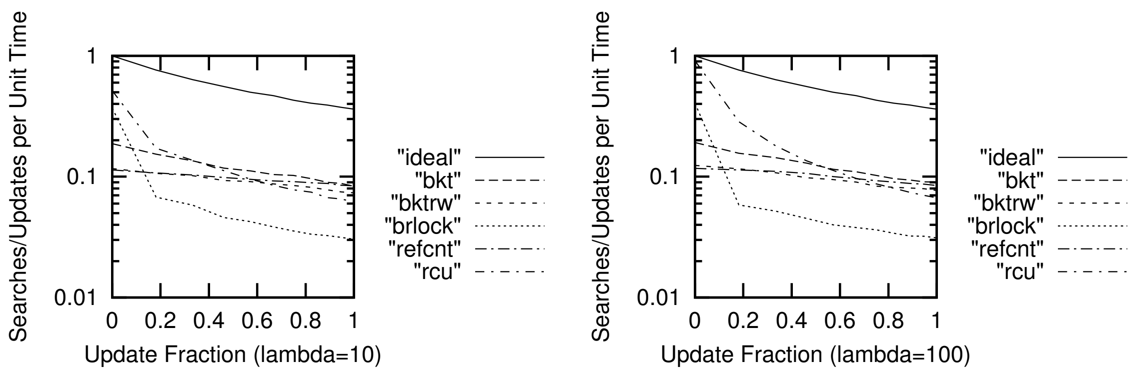

Figure 34: Opteron 2-CPU Performance for Mixed Workload

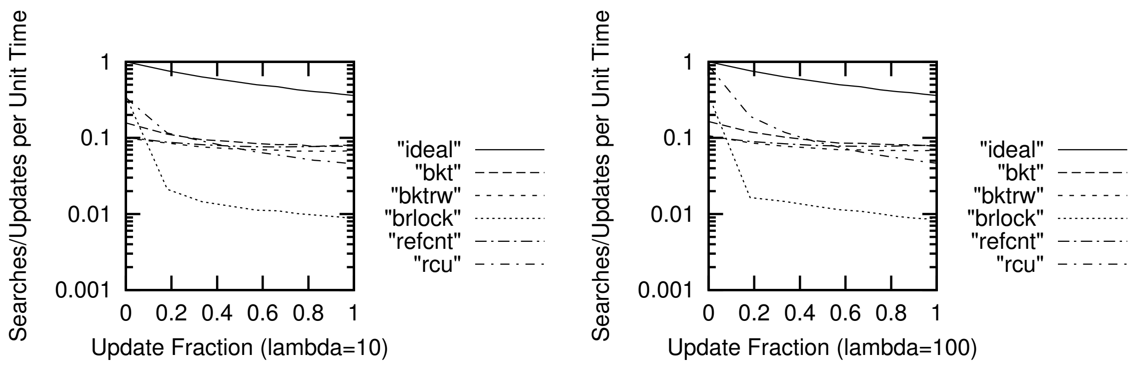

Figure 35: Opteron 4-CPU Performance for Mixed Workload

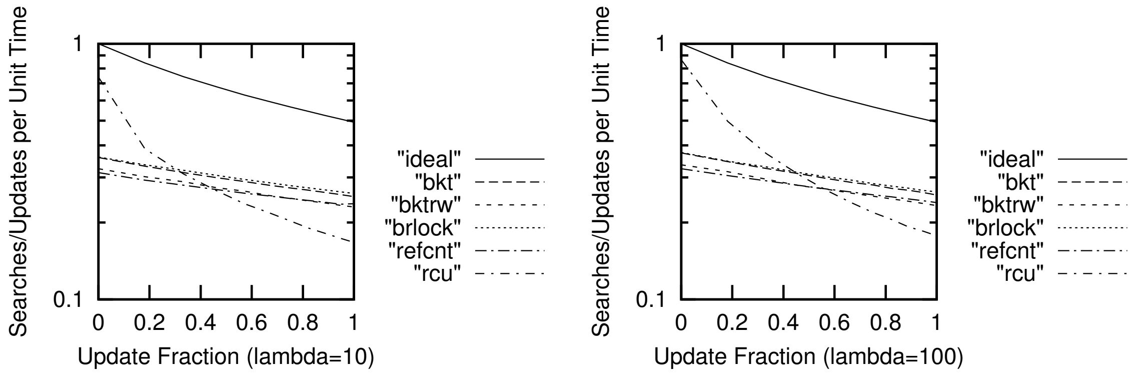

Figure 36: PPC 1-CPU Performance for Mixed Workload

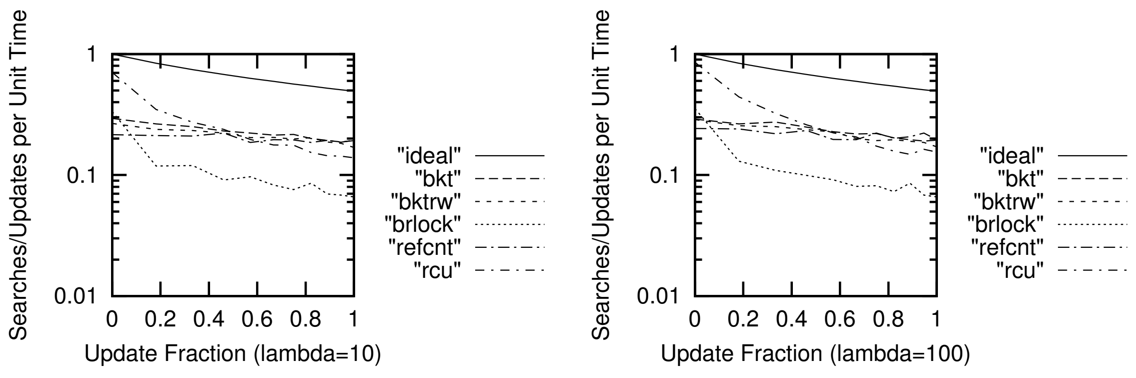

Figure 37: PPC 2-CPU Performance for Mixed Workload

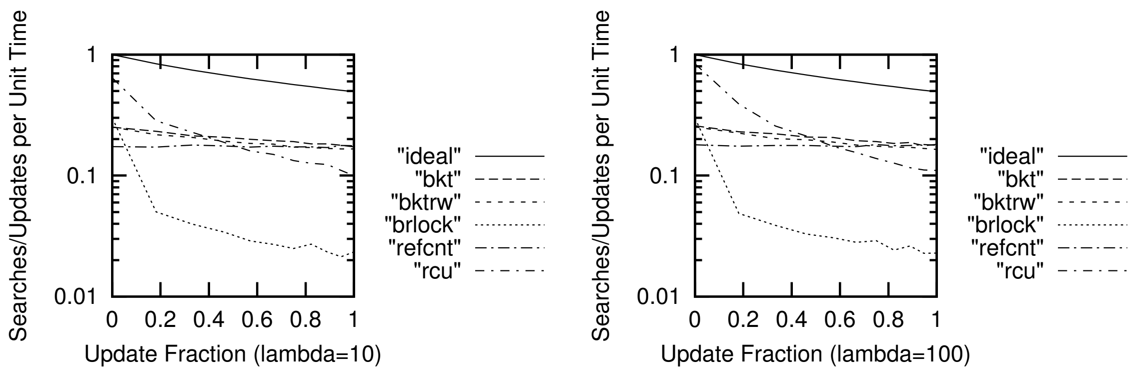

Figure 38: PPC 4-CPU Performance for Mixed Workload