Practical Concurrent Unrolled Linked Lists Using Lazy Synchronization

Kenneth Platz, Neeraj Mittal, and S. Venkatesan

The University of Texas at Dallas,
Richardson, TX 75080
{kplatz, neerajm, venky}@utdallas.edu

Abstract. Linked lists and other list-based sets are one of the most ubiquitous data structures in computer science. They are useful in their own right and are frequently used as building blocks in other data structures. A linked list can be "unrolled" to combine multiple keys in each node; this improves storage density and overall performance. This organization also allows an operation to skip over nodes which cannot contain a key of interest. This paper introduces a new high-performance concurrent unrolled linked list with a lazy synchronization strategy that allows wait-free read operations. Most write operations under this strategy can complete by locking a *single* node. Experiments show up to a 400% improvement over other concurrent list-based sets.

Keywords: concurrent data structures,lazy synchronization, linked lists

1 Introduction

In recent years, processor manufacturers have shifted their development focus away from increasing clock speeds and single-threaded performance. The rising prevalence of multi-core and multi-processor systems adds additional import to the quest for high-performance data structures that permit concurrent reads and writes while maintaining correct behavior.

Concurrent data structures can synchronize via several methods; the most common techniques involve locking. An exclusive lock can be used to control access to some portion of a data structure. When a thread attempts to access a portion of a data structure, it must first acquire one or more locks. Multiple techniques exist offering varying degrees of performance. The performance of a technique depends upon both the number of locks which must be acquired during an operation and the *granularity* of each lock (the fraction of the data structure protected by each lock).

Other algorithms use atomic read-modify-write instructions in lieu of locks. These instructions, such as compare-and-swap (CAS) or load-linked/store conditional (LL/SC), can be used to provide lock-free or wait-free synchronization [1]. Efficient lock-free and wait-free algorithms are inherently more complex than lock-based algorithms; they are harder to design, analyze, implement, and debug.

The linked list is one of the most ubiquitous data structures in computer science. It implements the standard set operations: *insert*, *remove*, and *lookup*. A linked list is typically implemented via a sequence of *nodes*, each of which contains a *key*, possibly a *data* element, and a pointer to the *next* node in the sequence. Linked lists are of particular interest because many other data structures (such as graphs and hash tables) use linked lists as "black box" subroutines [2].

Linked lists have been well-studied from a concurrency perspective. Several efficient lock-based algorithms exist. The simplest algorithm consists of a single lock which protects all accesses to the list, but this does not allow for any true concurrency. Improvements have been seen with fine-grained locking, where each node contains its own lock. These fine-grained algorithms typically scan the list for a node of interest and acquires the lock on that node(and possibly other nodes). Two algorithms that use this technique include an "optimistic" algorithm by Herlihy and Shavit [3] and a "lazy" algorithm by Heller [4].

Linked lists, while extremely useful, do have several disadvantages. One major disadvantage to a linked list is that any operation must, on average, traverse half the nodes in the list. Each step in this traversal must dereference that node's next pointer and access a memory location that may be far removed from the prior node. This access pattern makes poor use of the memory hierarchy found in today's systems.

Several attempts have been made to increase the efficiency of linked lists by combining multiple keys into a single node. These "unrolled" lists, first described by Shao et al [5], improve performance in two ways. First, these reduce the number of pointers which must be followed to find an item. Second, this groups multiple successive elements in sequential memory locations and better conforms to the principle of spatial locality [6, 7].

More recently Braginsky and Petrank developed a "chunked" lock-free linked list [8]. Their algorithm improves the locality of memory accesses by storing a sequential subset of key/data pairs within a contiguous block of memory. As time elapses and elements are inserted and removed from the list, their algorithm splits full chunks and combines sparsely populated ones. An operation can quickly locate the appropriate chunk, and searches within a chunk exhibit favorable spatial locality.

Our Contributions: We present a new lock-based data structure for an unrolled linked list based upon Heller's lazy synchronization wherein the majority of operations complete by locking a *single* node. We allow our data structure to contain up to K key/data pairs per node; this improves both the storage density and locality of reference within a node [6]. Using the algorithms we present, we can traverse this data structure in $O(n/K + K)$ operations, where n is the number of key/data pairs stored in the list. We also prove that our algorithms are correct, using linearizability as our safety property and deadlock-freedom as our liveness property.

The data structure we present is straightforward to implement and exhibits excellent throughput. Our analysis shows that it (i) exhibits high degrees of *spatial and temporal locality* by accessing sequential memory locations; (ii) re-

sponds extremely well to common *compiler optimizations*; and (iii) increases *cache efficiency* by eliminating extraneous pointers. In performance testing our implementation provides up to 400% higher throughput than the list presented by Braginsky and Petrank [8]; the improvement over several other concurrent lists is even higher.

Roadmap: The rest of the paper is as follows. Section 2 describes prior work related to this paper. Section 3 briefly describes our system model. Section 4 describes our data structure and the algorithms to implement standard set operations, while Section 5 provides a proof of correctness. Section 6 describes our experiments, and and Section 7 consists of our conclusions and suggestions for further work.

2 Related Work

Linked lists have been extensively studied in terms of concurrency; A number of lock-free and lock-based algorithms for linked lists exist, including lock-free algorithms by Valois [9], Michael [10], and Harris [11]. In this paper, we present a lock-based algorithm.

The data structure presented here is modeled after a list by Heller which uses a "lazy" locking strategy [4]. This implementation stores all keys in sorted order; a *scan* identifies the first key greater than or equal to the target key. The scan returns a *window* of two nodes: a node of interest and its immediate predecessor. A *lookup* operation returns TRUE if the key of the current node matches the key in question and FALSE otherwise. An *insert* or *remove* operation obtains a *window* from a scan operation and locks the predecessor and current nodes (in that order). The thread must next perform a *validate*; another thread may modify this section of the list before we acquire the locks. An *insert* then splices a new node into the list while a *remove* removes the node from the list.

Braginsky and Petrank recently developed a "chunked" lock-free linked list [8] which stores multiple sequential elements within the same memory block. This chunked list maintains chunk sizes within a specified minimum and maximum by splitting overfull chunks and merging underfull neighboring chunks. A merge or split requires "freezing" the chunk(s) in question to prevent further operations on a chunk. The operation must then *stabilize* the chunk to quiesce all pending operations. Multiple threads can help with the freeze and stabilize operations.

3 System Model

Our data structure implements a list-based set that supports three operations. A *lookup* operation accepts a *key* as an argument and returns either a *data* element indicating success or NIL indicating failure. An *insert* operation accepts a *key* and *data* element as arguments, returning TRUE if the operation successfully inserted the key/data pair or FALSE to indicate failure (due to the key already existing

in the list)[1]. A *remove* operation accepts a *key* and returns TRUE for success or FALSE if the element was not found.

The supporting algorithms use exclusive locks for coordination between threads. Many locking algorithms exist which provide different performance characteristics and progress guarantees. We assume a "black-box" lock which provides the guarantees of *deadlock-freedom* and *mutual exclusion*. For the sake of brevity, we rely on the Resource Acquisition Is Initialization(RAII) [12] idiom when acquiring these locks. We assume that acquiring a lock involves creating object with local scope which releases the lock when destroyed. In C++11, this is implemented with the `std::lock_guard` class. Other languages have similar constructs, either language-provided or user-specified.

4 An Unrolled Linked List Using Lazy Synchronization

4.1 Algorithm Overview

Our unrolled linked list maintains a singly-linked list of nodes and stores keys in partially sorted order. Each node contains (i) a *next* pointer to the next node in the list, (ii) a *marked* flag indicating a node's logical removal, (iii) an exclusive *lock*, (iv) a *count* of the number of elements in the node, and (v) an array of *key/data* pairs. In this data structure *lock* protects access to the *next* pointer which allows most operations to complete while holding a single lock. We define the parameter K to indicate the maximum number of key/data pairs per node and the *anchor* key as the first key in a node.

The data structure keeps track of the *head* pointer which points to the first element in the list. We maintain two invariants: (i) the anchor key of each node is strictly less than the anchor key of its successors and (ii) all (non-anchor) keys in a node are strictly greater than their anchor key We do not impose any ordering among keys within a node; attempting to keep keys in sorted order would penalize write performance and complicate wait-free lookups. The layout of the data structure is depicted in Fig. 1.

We define two sentinel values of $-\infty$ and $+\infty$. We initialize the list with three sentinel nodes with anchor keys of $-\infty$, $-\infty$, and $+\infty$. The sentinel value of $\top$ indicates a key slot that is unused.

Each operation scans the list to find the correct nodes upon which to operate and returns that node and its predecessor. An insert replaces a sentinel key with our new key/data pair and returns TRUE if successful or FALSE if the element is already in the list. A remove replaces a key with a sentinel key, returning TRUE for success or FALSE for failure (i.e. the element was not found in the list). A lookup returns either the *data* element associated with the key or NIL if the key is not in the list.

[1] Sets do not permit multiple entries for a given key. Another option is to replace the existing data element with the new element.

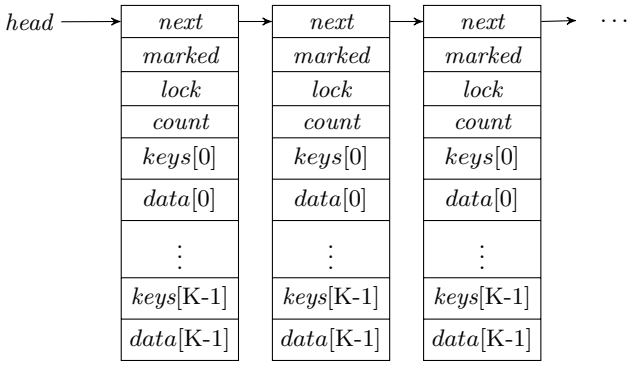

Fig. 1. Layout of the unrolled linked list

4.2 Algorithm Detail

The first step in any operation on the list involves a *scan* (Alg. 1). We maintain three pointers during this scan, *pred*, *curr*, and *succ*. We scan through the list until the *succ* contains an anchor key greater than our key of interest Once *succ* meets this criteria, *scan* returns the pair (*pred*, *curr*). We also assume the existence of a *contains* function. This function takes an *array* and an *item* as arguments and returns either the location of the first occurrence of *item* in the array or NIL[2].

```
1 Function scan(item) : (node,node)
2     prev ← head
3     curr ← prev.next
4     succ ← curr.next
5     while succ.key > item do
6         prev ← curr
7         curr ← succ
8         succ ← succ.next
9     return (prev,curr)
```

Alg. 1: Scan

The *lookup* function starts with a *scan* of the list(Alg. 2). Once we have our *curr* node, we perform a single pass through its keys looking for *item*. At each slot, we read the key/data pair *atomically* (line 13). We can either select key and data elements that collectively fit within a machine word or use atomic snapshots [13, 14]. All of our tested implementations used the former technique[3]

[2] We can use `std::find_first_of`(C++) or `Collection.contains`(Java).
[3] Specifically we store a 32-bit key and data element in a 64-bit word.

[8]. If we encounter *key* during our scan, we return the associated *data* element. If we reach the end of the keys without finding *item*, it may still be present; a *remove* operation may relocate *item* to the anchor key of a node. In this case, we must re-read the anchor key/data pair. If *item* is present, we return its associated *data* element; otherwise, we return NIL.

```
10 Function lookup(item) : Data
11     (prev, curr) ← scan(item)
12     for i ← 0 to K − 1 do
13         (key, data) ← (curr.keys[i], curr.data[i])
14         if key = item then return data
15     (key, data) ← (curr.keys[0], curr.data[0])
16     if key = item then return data
17     return NIL
```

Alg. 2: Lookup

Our *insert* and *remove* functions depend upon a *validate* function (Alg. 3) similar to Heller's [4]. We must perform this validation because another thread may still manipulate *prev* or *curr* until we acquire the lock on *prev*. We validate by checking that neither *prev* nor *curr* are marked for removal, *prev.next* still points to *curr*, and our target key is not less than *curr's* anchor key[4].

```
18 Function validate(prev, curr, item) : boolean
19     return ¬prev.marked ∧ ¬curr.marked
       ∧prev.next = curr ∧ curr.keys[0] <= item
```

Alg. 3: Validate

The *insert* operation (Alg. 4), performs a *scan* to locate an appropriate insertion point, locks *prev*, performs a *validate*. If the validation fails, it must re-scan. Once a validation succeeds, it checks *curr* for three conditions. If *curr* already contains *item* it leaves the node unchanged and returns *false*. If there is at least one empty slot in *curr* (denoted by the sentinel $\top$) it atomically replaces the sentinel key and its associated data element with the new key/data pair, increments *count*, and returns *true*. If there are no available slots it must split the node. To split a node (Alg. 5), we first lock *curr*. This will not require another validation since no other thread can modify *prev.next*. Next we allocate two new nodes, *new1* and *new2*. We copy all of the key/data pairs from *curr* to *new1*,

[4] A concurrent removal of *curr*'s anchor key may result in this violation.

```
20 Function insert(key,data) : boolean
21     while true do
22         (prev,curr) ← scan(item)
23         prev.lock()
24         if ¬validate(prev,curr,item) then
25             continue                    /* Return to head and re-scan */
26         if curr.contains(item) then return FALSE
27         slot ← first location of ⊤ in curr
28         if slot is defined then
29             (curr.keys[slot], curr.data[slot]) ← (key, data)
30             curr.count ← curr.count + 1
31         else
32             curr.lock()
33             (new1, new2) ← split(curr)
34             if key < new2's anchor key then
35                 (new1.keys[K/2], new1.data[K/2]) ← (key, data)
36                 new1.count ← new1.count + 1
37             else
38                 (new2.keys[K/2], new2.data[K/2]) ← (key, data)
39                 new2.count ← new2.count + 1
40             curr.marked ← true
41             prev.next ← node1
42         return TRUE
```

Alg. 4: Insert

sort them, and then copy the upper half to *new2*[5]. Finally, we replace the upper half of *new1*'s keys with $\top$.

```
43 Function split(node) : (Node, Node)
44     Allocate two new nodes, new1 and new2
45     Copy all key/data pairs from node to new1
46     Sort all key/data pairs in new1 ascending by key
47     Copy the upper ⌊K/2⌋ key/data pairs from new1 to new2
48     Replace the upper ⌊K/2⌋ keys in new1 with ⊤
49     new1.next ← new2
50     new2.next ← node.next
51     new1.count ← ⌈K/2⌉, new2.count ← ⌊K/2⌋
52     return (new1, new2)
```

Alg. 5: Split

[5] While there is a $O(n)$ algorithm to determine the median and partition a set of values, in real-world situations, an efficient sorting algorithm is faster [2].

```
53 Function remove(item) : boolean
54     while true do
55         (prev, curr) ← scan(item)
56         prev.lock()
57         if ¬validate(prev, curr,item) then
58             continue
59         slot = curr.contains(item)
60         if slot is not defined then
61             return FALSE
62         if slot = 0 then
63             min ← location of next-smallest key in curr
64             (curr.keys[0], curr.data[0]) ← (curr.keys[min], curr.data[min]
65             curr.keys[min] ← ⊤
66         else
67             curr.keys[slot] ← ⊤
68         curr.count ← curr.count − 1
69         if curr.count <MinFull then
70             curr.lock()
71             succ ← curr.next
72             if succ.keys[0] = +∞ then
73                 return TRUE
74             if curr.count = 0 then
75                 curr.marked ← true
76                 prev.next ← succ
77                 return TRUE
78             succ.lock()
79             if curr.count + succ.count < MaxMerge then
80                 merge(curr, succ)
81             else
82                 (new1, new2) ← rebalance(curr, succ)
83                 prev.next ← new1
84         return TRUE
```

Alg. 6: Remove

Removing an element operates in a similar manner (Alg. 6). We perform a *scan* to locate the (*prev*, *curr*) pair, lock *prev*, and *validate*. If this succeeds, we attempt to locate *item* in *curr.keys*. If it is not present, we return FALSE. If *item* is present but not the anchor key, we replace *item* with the sentinel $\top$. If *item* is the anchor key, we locate the next-smallest key. We then replace the anchor key/data pair with the next-smallest key/data pair and replace the next-smallest key with $\top$. At this time, we also decrement the node's *count*. If our node now has fewer than MinFull keys, some additional checking is required. We cannot merge with the tail (line 74), and if our node is empty (line 72) we delete it

outright[6]. Otherwise, we either merge with our successor node(Alg. 7) or create two new nodes and partition the key/data pairs equally among them(Alg. 8).

```
85 Function merge(curr, succ)
86     Copy valid key/data pairs from succ to curr
87     succ.marked ← TRUE
88     curr.next ← succ.next
```

Alg. 7: Merge

```
89 Function rebalance(curr, succ) : (Node, Node)
90     Create two new nodes new1 and new2
91     Copy valid key/data pairs from curr and succ to new1
92     Sort all key/data pairs in new1 by ascending key value
93     Copy the upper ⌊K/2⌋ key/data pairs from new1 to new2
94     Replace the upper ⌊K/2⌋ keys in new1 with ⊤
95     new2.next ← succ.next, new1.next ← new2
96     return (new1, new2)
```

Alg. 8: Rebalance

Optimization We can further modify the above algorithms to keep all valid keys at the head of the node; this requires minor changes to *remove*. Instead of replacing the affected key with $\top$, wewould replace that key with the last valid key in the node and replace the last valid key with $\top$ (this is symmetric to removing the anchor key). Likewise, the *lookup* function would scan right-to-left, starting with the last valid key.

5 Proof of Correctness

For the purposes of the correctness proof, we assume that the memory of nodes that have become garbage is not reclaimed. We further assume that the key space is finite.

We will make use of the following definitions in this proof: an *active node* is reachable from the head of the linked list; a *passive node* was an active node, but is no longer active; and the *target node* is the node upon which an operation performs its action. Note that the address of the target node will be stored in *curr* when a traversal terminates.

The following propositions about our algorithm can be easily verified.

[6] This can happen if *succ* is the tail.

Proposition 1. *The target node returned by a traversal is active at some point during the traversal.*

Proposition 2. *Insert and delete operations only operate on active nodes.*

Proposition 3. *The anchor key value of a node can never decrease.*

Proposition 4. *Consider two nodes x and y such that the next field of x points to y. Then, the anchor key value of x is strictly less than the anchor key value of y.*

We are now ready to prove the correctness of our algorithm.

5.1 All Executions are Linearizable

We show that an arbitrary execution of our algorithm is linearizable by specifying the *linearization point* of each opearation [20]. Note that the linearization point of an operation is the point during its execution at which the operation appears to take effect. Our algorithm supports three types of operations: *lookup*, *insert* and *remove*. We will subdivide our *search* operations into two types: *lookup-hit* (if the operation finds its target) and *lookup-miss*(otherwise). We can also, for the sake of convenience, treat *insert* and *remove* operations which do not modify the data structure as *lookup-hit* and *lookup-miss* operations, respectively.

We now specify the linearization point of each operation.

1. *Insert operation:* There are two cases depending on whether the operation performs a split action on the target node or not. If it does not perform a split, then the operation's linearization point is defined as the point at which it copies its target key to an empty slot in the target node. Otherwise, it is defined as the point at which it updates the *next* field of the *pred* node.
2. *Remove operation:* The operation's linearization point is defined to be the point at which it replaces the slot in the target node containing its target key with a sentinel value.
3. *Lookup-hit operation:* There are two cases depending on whether the target node was active when the operation read the contents of the slot containing its target key. If the target node was not active, then the operation's linearization point is taken to be the point at which the target node became passive. Otherwise, it is taken to be the point at which it read the contents of the slot.
4. *Lookup-miss operation:* There are two cases depending on whether the target node was active when the operation when the operation finished scanning the target node. If the target node was not active, then the operation's linearization point is taken to be the point at which the target node became passive. Otherwise, it is taken to be the point at which it started scanning the target node.

The following lemma can now be easily verified:

Lemma 1. *Consider an arbitrary execution of our algorithm. The sequence of operations obtained by ordering operations based on their linearization points is legal, that is, all operations in the sequence satisfy their specification.*

The above lemma implies that:

Theorem 1. *Every execution of our algorithm is linearizable.*

5.2 All Executions are Deadlock-Free

Using Proposition 4, it can be easily verified that any operation will encounter increasing larger anchor key values while traversing the list. Thus we have:

Lemma 2. *Any traversal of the linked list terminates eventually.*

For convenience, we refer to insert and remove operations as *modify* operations.

Definition 1 (quiescent state). *The system is in a* quiescent state *if no modify operation completes hereafter.*

Definition 2 (potent state). *The system is in a* potent state *if it has one or more pending modify operations.*

Note that a quiescence is a *stable* condition; once the system is in a quiescent state, it stays in a quiescent state. We show that our algorithm is deadlock-free by proving that a potent state is necessarily non-quiescent. Assume, by the way of contradiction, that there is an execution of the system in which the system eventually reaches a state that is potent as well as quiescent. The following propositions about our algorithm can be easily verified.

Proposition 5. *Once the system has reached a quiescent state, the linked list cannot change any more.*

Proposition 6. *If the system is in a quiescent state, then no active node can be marked.*

Intuitively, the above proposition holds because a modify operation does not acquire any locks after it starts marking nodes. As a result, such a modify operation will eventually complete. This is, in turn, implies that this system is not in a quiescent state—a contradiction

The following lemmas can be easily proved using the above proposition:

Lemma 3. *Any validation test invoked by a modify operation that started its traversal after the system reached a quiescent state will succeed.*

Proof. This follows from Proposition 5. A validation test will only fail if the list changes in the immediate vicinity of *curr*. □

Lemma 4. *Once the system has reached a quiescent state, a modify operation can restart at most once thereafter.*

Proof. (By contradiction). Suppose not. Let us assume, for the sake of contradiction, that the system enters a quiescent state at time t_q. Let us further define m_q as the last modify operation completed prior to the system entering the quiescent state.[7] Let us also define a modify operation m' such that $invoke(m') < t_q$, but m' restarts its scan at times t' and t'' ($t', t'' > t_q$). How can this occur?

Suppose that m_q marked at least 1 node during its course of execution. Since $invoke(m') < t_q$, it is possible that at $t' > t_q$, m' may need to restart its *scan* due to the aforementioned marked node. However, Lemma 3 established that the any validation test invoked by this scan will succeed. Therefore, the scan will not need to restart and our original assumption must be false. □

Lemma 5. *If the system is in a quiescent state, then the rightmost modify operation will successfully acquire all the locks it needs.*

Proof. Recall that our modify operations acquire locks in head-to-tail fashion. If we are in a quiescent state, we can look at modify operation operating on the set of nodes closest to the tail of the list. Since there is no modify operation accessing nodes closer to the tail, this operation will eventually succeed. □

The above lemma implies that if the system is in a state that is both potent and quiescent, then the rightmost modify operation will eventually complete. This contradicts the assumption that the system is in a quiescent state. Thus we have:

Theorem 2. *Every execution of our algorithm is deadlock-free.*

6 Experimental Evaluation

6.1 Experiment Setup

We completed our experiments on a 2-processor AMD Opteron 6180SE system with a clock speed of 2.5GHz, 24 total execution cores, and 64GB of memory running Linux (kernel 2.6.43). All of our evaluation code was written in C++ and compiled using gcc-4.8.3 using the same set of optimizations(-O3 -funroll-loops -faggressive-loop-optimizations -fprefetch-loop-arrays). We evaluated the following list implementations:

1. **Lazy:** The lazy linked list by Heller [4].
2. **LockFree:** A lock-free linked list by Harris [11] & Michael [10, 16].
3. **Chunked:** The chunked linked list by Braginsky and Petrank [8][8].
4. **Unrolled:** The unrolled linked list described in this paper.

[7] Since we have proved that our algorithm is linearizable, we can therefore define an order of operations based upon their linearization points.

[8] Source code was obtained with permission from Braginsky and Petrank.

Each implementation used hazard pointers for garbage collection. For our initial experiments we tested node sizes ranging from 8 to 512 keys per node, key ranges from 1,024 to 1 million, thread counts ranging from 1 to 48, and multiple workload mixes. Based on our initial observations, we feel the following parameters accurately represent the performance of our and other list implementations[9]:

1. **Node Size:** For the chunked and unrolled lists, we evaluated the performance with K of 8 and 500 keys per node, MinFull of $K/4$ and MaxMerge of $3K/4$.
2. **Workload Distribution:** We evaluated performance against three representative workloads: write-dominant with no lookups, 50% inserts, 50% removes; balanced with 70% lookups, 20% inserts, 10% removes; and read-biased with 90% lookups, 9% inserts, 1% removes.
3. **Degree of Concurrency:** We evaluated the performance with 1, 2, 4, 8, 12, 16, 20, and 24 threads.
4. **Key Range:** Keys were allowed to range from 0 to 65,535 (inclusive)

Each experiment was conducted initially creating a list with 10,000 entries. We then spawned the specified number of threads and ran them concurrently for 8 seconds. Each thread executed as many operations as possible using the specified mix of operations, and we recorded the total number of operations. Each experiment was repeated until we achieved a 95% confidence interval less than 10% of the mean. All results are reported in operations per second.

6.2 Experimental Results

Fig. 2 depicts the results of our experiments. The graphs on the left depict results for 8 keys per node while those on the right show 512 keys per node. From top to bottom we display results for the 0/50/50, 70/20/10, and 90/9/1 workloads, respectively. These results show that the relative performance of each algorithm remains consistent for every workload and thread count. Specifically, we can rank them fastest to slowest: our unrolled algorithm, Braginsky and Petrank's chunked algorithm, Harris and Michael's lock-free algorithm, and Heller's lazy algorithm. The relative performance at 24 threads is shown in Table 1.

			K = 8		K = 512	
Workload	Lazy	Lock-Free	Chunked	Unrolled	Chunked	Unrolled
0/50/50	100	132	265	621	1674	17826
70/20/10	100	126	245	448	4091	18847
90/9/1	100	121	232	383	3599	12994

Table 1. Relative performance at 24 threads with respect to the Lazy algorithm

[9] Additional experimental data is available in the companion technical report [15].

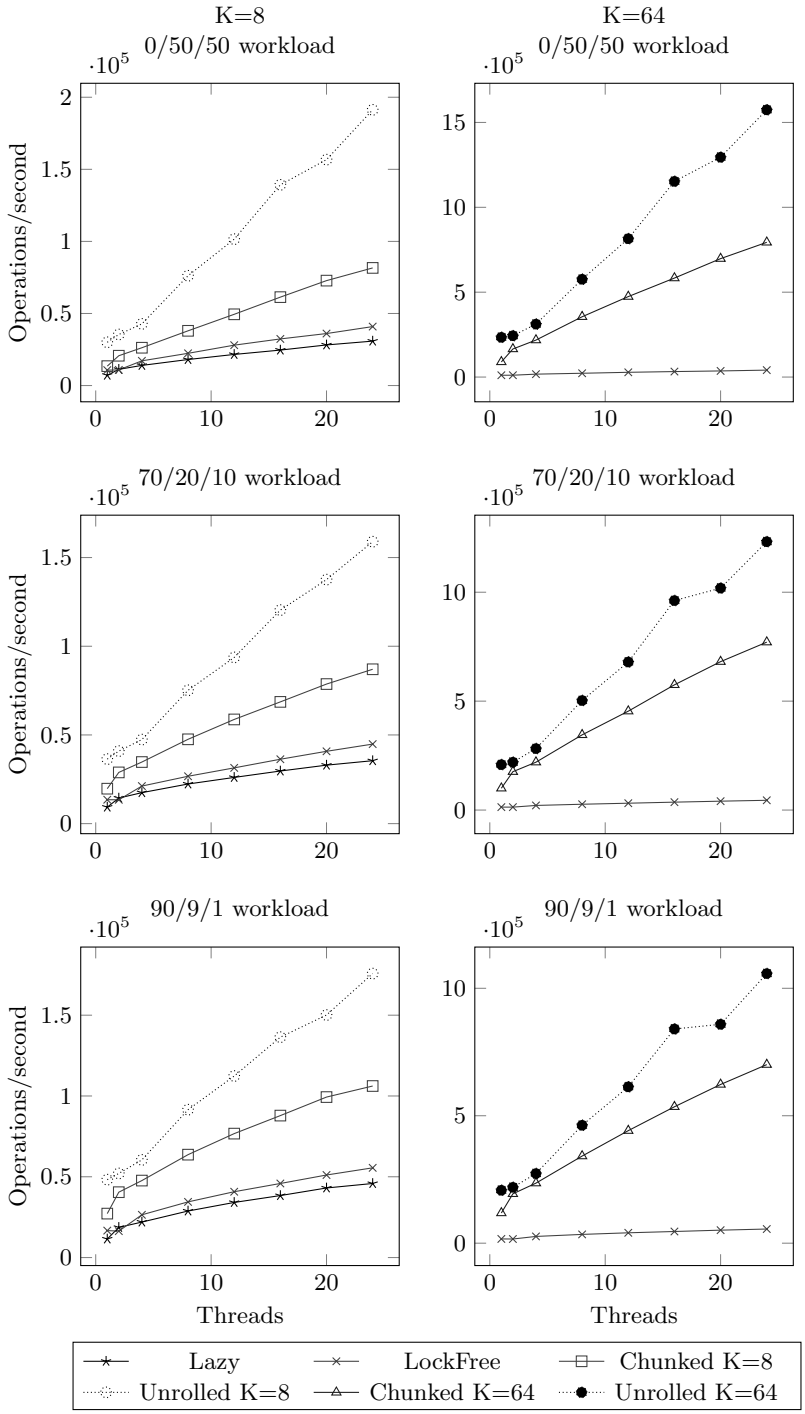

Fig. 2. Experimental Results

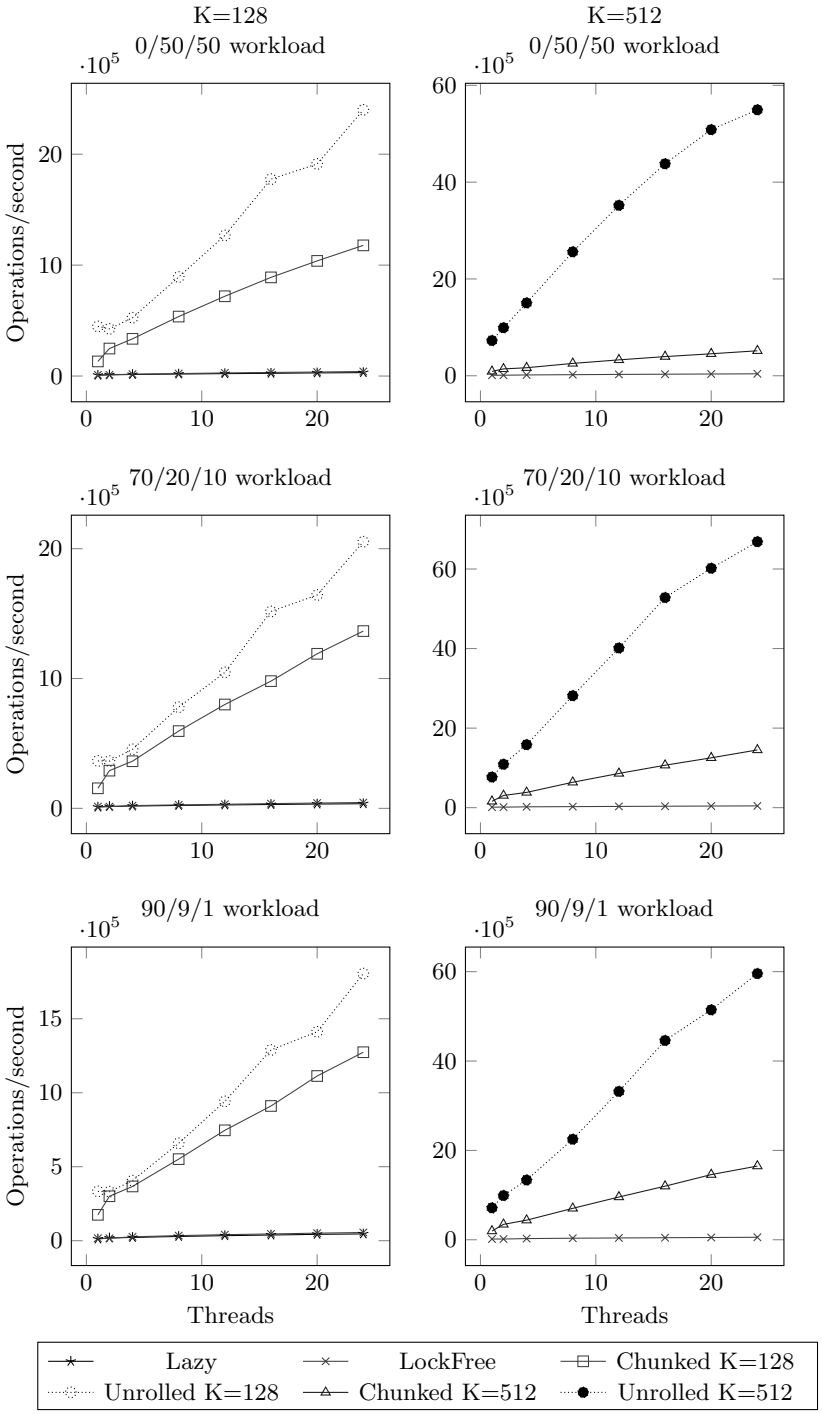

Fig. 3. Experimental Results

Intuitively we can divide our algorithms into two groups. The lazy and lock-free lists operate as a single phase; each step along the list visits each entry in turn. The chunked list and unrolled list take a two phase approach. The first phase skips over multiple entries while seeking the correct node, and the second phase scans sequential entries in that node.

When we consider the performance differential between our data structure and the chunked list, we should consider the following substantial differences:

1. A lookup in the chunked list (once the node has been identified) involves repeatedly dereferencing a pointer and accessing a different area of the chunk. Our list scans sequentially through an array. Compilers can aggressively optimize array scans using techniques such as loop unrolling, cache prefetching and software pipelining [17]. A compiler can (in some cases) also use vector instructions to perform multiple comparisons concurrently.
2. In order to perform a split or rebalance, the chunked list must first freeze and stabilize the affected node(s). Freezing requires visiting each entry and setting a freeze bit (using CAS) while stabilizing involves traversing the chunk and removing any partially-deleted nodes. Our list only requires two calls to the *copy* library routine and one call to *sort* to perform either an insert or remove. These library routines are typically aggressively optimized for performance.
3. Once a node has been located within the list, any operation in our list involves a sequential scan through its memory. This provides the added benefit of *spatial locality* [18]. When we access a key/data pair, it is likely on the same cache line as the last-accessed pair.[10]

In order to measure the effect of compiler optimizations on our list and the chunked list, we disabled all optimizations and recompiled. We then reran our experiments using the 70/20/10 "balanced" workload with 8 and 500 keys per node. We then reported the speedup percentage for each degree of concurrency(Fig. 4).

The results confirm our hypothesis. While the chunked list did see moderate improvements (35%) at $K = 8$ and significant (200%) improvement at $K = 512$, our list achieved a minimum of a 400% improvement for each thread count. We saw the greatest improvement, 784%, with $K = 512$ and 2 threads.

7 Conclusions and Future Work

Braginsky and Petrank described a means to reorganize a linked list to improve locality of memory access; in this paper we have improved upon their algorithms. By storing multiple keys in a node and skipping irrelevant nodes, we can improve performance by a constant factor over traditional linked lists. Storing the entries in an unsorted array allows us to sequentially scan these entries, a task which

[10] On the AMD architecture we tested, a key/data pair consumes 8 bytes, and the cache line stores 64 bytes. This allows 8 key/data pairs to share a cache line.

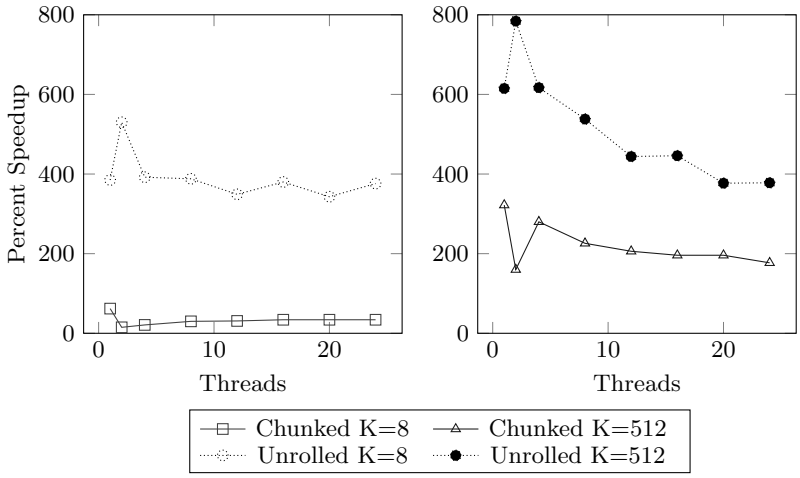

Fig. 4. Optimization Impact

compilers can aggressively and effectively optimize. Our results are extremely encouraging and suggest that further research should be done in this area.

We envision three different ideas for further research. One possible improvement involves the use of a *group mutual exclusion* object to control access to a node [19]; this would permit multiple *insert* or multiple *remove* operations to operate on the same node concurrently. Second, we can develop a lock-free implementation of this object. We would expect either technique to provide an incremental performance improvement over what we have presented. Finally, we would like to explore the effects of using our data structure as a "black box" subroutine to construct more complex data structures.

References

1. Herlihy, M., Shavit, N.: On the nature of progress. In Fernndez Anta, A., Lipari, G., Roy, M., eds.: OPODIS'11. Volume 7109 of LNCS. Springer Berlin Heidelberg (2011) 313–328
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. 3rd edn. The MIT Press (2009)
3. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. revised 1st edn. Elsevier, Inc, Waltham, Massachussetts (2012)
4. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. In: OPODIS'05. LNCS, Berlin, Heidelberg, Springer-Verlag (2006) 3–16
5. Shao, Z., Reppy, J.H., Appel, A.W.: Unrolling lists. In: LFP'94, New York, NY, USA, ACM (1994) 185–195
6. Demaine, E.: Cache-oblivious algorithms and data structures. Lecture Notes from the EEF Summer School on Massive Data Sets (2002)

7. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design: The Hardware/Software Interface. 5th edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2013)
8. Braginsky, A., Petrank, E.: Locality-conscious lock-free linked lists. In Aguilera, M., Yu, H., Vaidya, N., Srinivasan, V., Choudhury, R., eds.: ICDCN'11. Volume 6522 of LNCS. Springer Berlin Heidelberg (2011) 107–118
9. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: PODC'95, New York, NY, USA, ACM (1995) 214–222
10. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA'02. LNCS, New York, NY, USA, ACM (2002) 73–82
11. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: DISC'01. LNCS, London, UK, UK, Springer-Verlag (2001) 300–314
12. Stroustrup, B.: The Design and Evolution of C++. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1994)
13. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM **40** (1993) 873–890
14. Anderson, J.H.: Composite registers. In: Distributed Computing. (1993) 15–30
15. Platz, K., Mittal, N., Venkatesan, S.: Practical concurrent unrolled linked lists using lazy synchronization. Technical Report UTDCS-09-14, The University of Texas at Dallas Computer Science Department (2014)
16. Michael, M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Transactions on Parallel and Distributed Systems **15** (2004) 491–504
17. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
18. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. 5th edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2011)
19. Joung, Y.J.: Asynchronous group mutual exclusion (extended abstract). In: PODC '98, New York, NY, USA, ACM (1998) 51–60
20. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12** (1990) 463–492