Restricted Transactional Memory and SkipLists

Connor Lightsey

Brown University, Computer Science Department, Providence, RI 02912

I. Introduction

A thread cannot change the value of a variable in memory automatically. This is normally not an issue, as by the time the code attempts to reference the variable after it has been created the value of that variable has been set. When running a program with multiple threads in parallel, however, this becomes a potential problem, as two threads interacting on the same variable can lead to unpredictable results. To prevent this from happening, a thread must indicate to other threads that the variable is being changed, and "lock" the variable so that no other thread can access it while the variable is being updated. The portion of the code which is locked is known as the "critical section". Ideally, the size of the critical section should be as small as possible, as the larger the critical section is the longer a thread must hold onto the lock and the longer other threads must wait for the lock to be released [1]. In addition, it is possible for threads to delete variables before other threads are done with them. To prevent this, pointers to the deleted nodes, known as hazard pointers, are temporarily saved until the delete operation is fully complete.

For the purpose of this thesis, various locking techniques were investigated on a SkipList. A SkipList is a list made of multiple layers, each layer more sparse than the layer below, as shown in Figure 1. The bottom layer will always contain every element in the list, and there is a fixed percentage chance that each element will be added to subsequently higher layers. This allows the list to be traversed faster than a normal list, as the program can look for a value at a higher layer until it passes the value, then proceed to a lower level and continue the

search. This approach results in a list which is very efficient in searching, but also very costly in adding and removing.

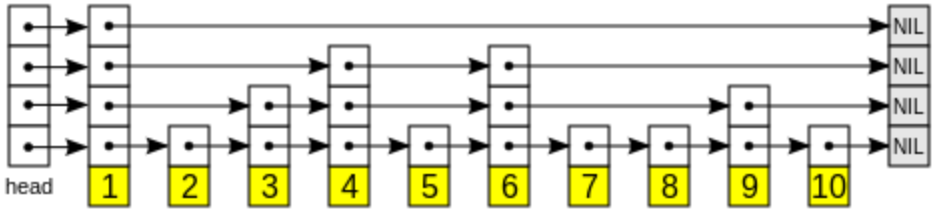

Figure 1. Example SkipList implementation with multiple layers [2].

II. SkipList Methods

Four various types of SkipLists were investigated for code locking, which are described below.

A. Coarse SkipList

In a SkipList, the critical section is the pat of the program that searches for the actual element by either adding, removing, or checking to see if the list contains that element. A very naive approach would be to simply lock the head of the list before beginning the search process and then unlocking the head of the list after completing the operation. While this does ensure that the threads will never overlap, it also prevents the code from executing in parallel at all. Since this is not incredibly finely tuned and will also include basic Hazard Pointer management, this approach shall be referred to as SkipListCZ, or SkipList Coarse Hazard-Pointer Management.

B. Hand-Over-Hand SkipList

A more finely-tuned locking method uses hand-over-hand locking. In this method, the thread attempts to lock a node beyond the node it is currently at and, if successful, releases the previous node's lock [3]. This ensures that no other thread will affect the portion of the list that is being locked, as other threads attempting to access the same section of code will encounter

the lock and will be unable to proceed. However, it must hold the lock at every level before it can actually interact with the data, and cannot unlock any higher level until it completes its interaction with the SkipList. Due to the nature of the locking approach and using basic Hazard Pointer Management as well, this will be referred to as SkipListHZ, or SkipList Hand-Over-Hand Hazard-Pointer Management.

Transactional Memory

If memory could be changed automatically, or “atomically”, this would greatly reduce the amount of locking necessary. Based on this idea, transactional memory was created in order to allow memory change commands, also known as transactions, to execute seemingly atomically. Restricted Transactional Memory, or RTM, is Intel’s code which allows the program to access this functionality [4]. RTM works by storing the commands into memory, and then completes the commands in a way that appears atomic to all other threads.

Since it is storing the commands in memory, it is possible that the transaction will abort attempting to store too many commands. In addition to potentially aborting the transaction if it requires too much memory, the transaction will also abort if the thread attempts to use any banned operation. One such operation is the compare and swap operation, which is typically used to acquire and release locks. This means that any thread using RTM must either use a different method for acquiring and releasing locks within a transaction or simply acquire and release the lock outside of the transaction. For the sake of this thesis, anything contained within a transaction shall be referred to as a teleportation.

C. Coarse Teleporting SkipList

In the SkipList, the threads use transactional memory to traverse through the list. This means that not much will change in the case of the coarse lock, since the thread will not attempt to acquire a lock within the transaction. Therefore, the Coarse Teleporting SkipList, or

SkipListCTZ, will be the same as the SkipListCZ but the list traversal will be contained within a transaction.

D. Hand-Over-Hand Teleporting SkipList

As mentioned before, hand-over-hand locking acquires and releases locks as it proceeds through the list. As such, for the hand-over-hand lock to work, the thread must acquire and release the locks within the list traversal, which is in this interpretation within the teleportation. However, the thread must use a different method to acquire locks, as any lock acquired using the compare and swap method within a transaction would abort the transaction. To fix this problem, the thread will first check to see if the code is in a transaction when attempting to acquire a lock. If it is in a transaction, the thread then checks if the lock of the next node in the list is already owned, and if so aborts the transaction. Otherwise, the thread acquires the lock. If the thread is not in a transaction, the lock proceeds as normal. Besides this change and a transaction containing the list traversal, the Hand-Over-Hand Teleporting SkipList, or SkipListHTZ, should be the same as the SkipListHZ.

III. Experimental Results

To test the efficiency of the SkipLists at various operations, a driver was used which runs the SkipList through a fixed number of operations with a percentage of those operations mutating, or adding elements to or removing elements from, the SkipList. The rest of the operations checked to see if the SkipList contained various elements. The efficiency of the SkipLists were determined based on the amount of runtime it took for the SkipList to complete these operations. The runtime is recorded within the driver using gethrtime [5].

A. Expectations

Since the hand-over-hand SkipLists are more finely tuned than the coarse SkipLists, both SkipListHZ and SkipListHTZ should perform better than SkipListCZ and SkipListCTZ. Since the teleportation only affects the list traversal, both teleporting versions should have comparable runtimes as their non-teleporting versions. In addition, the higher the percentage of mutations, the more time the SkipList will take due to the high cost on adding and removing from the SkipList. Adding more threads should decrease the runtime of the hand-over-hand SkipLists but should have little to no effect on the coarse SkipLists. Finally, the SkipLists which implement teleportation should scale better with more operations than the SkipLists which do not implement teleportation due to faster list traversal.

B. Test Results

Initially, the SkipLists were run with 4 threads over 100,000 operations with 25%, 50%, and 75% mutations. After running the four different types of lists, the measured trends are presented in Figure 2.

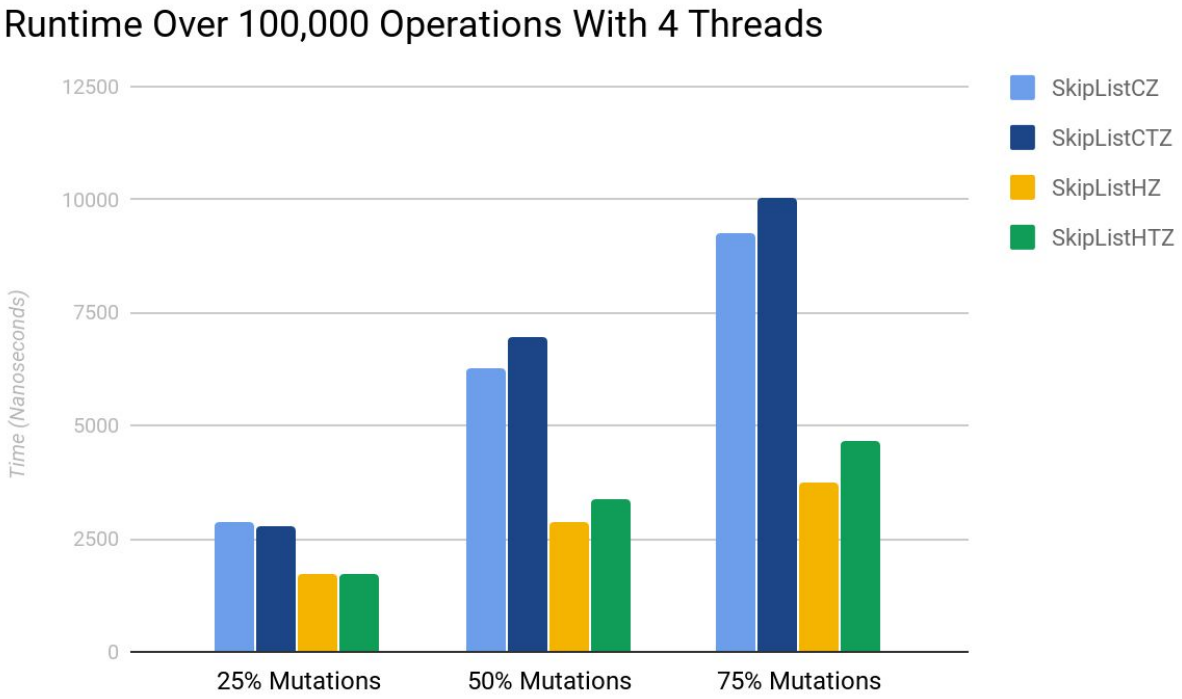

Figure 2. Measured SkipList code runtime versus mutations frequency.

As expected, the Hand-Over-Hand SkipLists performed consistently better than the Coarse SkipLists. In addition, the teleporting SkipLists performed comparably to their non-teleporting versions, although they were consistently worse in all but the 25% mutation case. Teleportation was very successful in the case of SkipListCTZ, which had an average teleportation distance of 141 nodes at 25%, 120 nodes at 50%, and 103 nodes at 75% mutations. However, teleportation was very unsuccessful in the case of SkipListHTZ, which failed the majority of times and had an average teleportation distance of 18 nodes at 25%, 20 nodes at 50%, and 22 nodes at 75% mutations.

Next, the four SkipLists were run over 100,000 operations with 25% mutations with 1, 2, and 4 threads. The measured runtime results are presented in Figure 3.

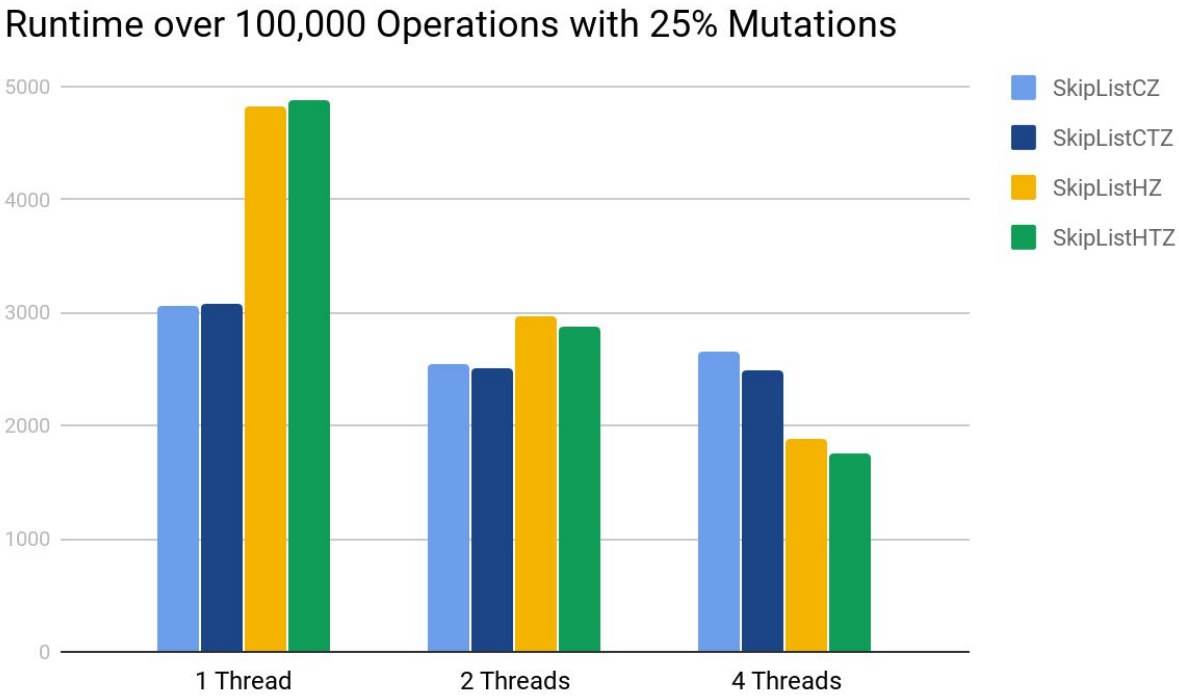

Figure 3. Measured SkipList code runtime versus number of threads.

In this scenario, there was a very slight increase in performance in the coarse SkipLists, but not nearly as dramatic as the change when increasing the number of threads in the hand-over-hand SkipList. In addition, there was a consistently high successful transaction rate on both SkipListCTZ and SkipListHTZ. There was an average teleport distance of 275 nodes for 1 thread, 202 nodes for 2 threads, 150 nodes for 4 threads on SkipListCTZ. Finally, on SkipListHTZ there was an average teleport distance of 29 nodes for 1 thread, 24 nodes for 2 threads, and 18 nodes for 4 threads.

Finally, the SkipLists were tested to see how increasing the number of operations affects performance. The SkipLists were run with 4 threads and 25% mutations over 50,000, 100,000, and 500,000 operations, giving the results presented in Figures 4, 5, and 6.

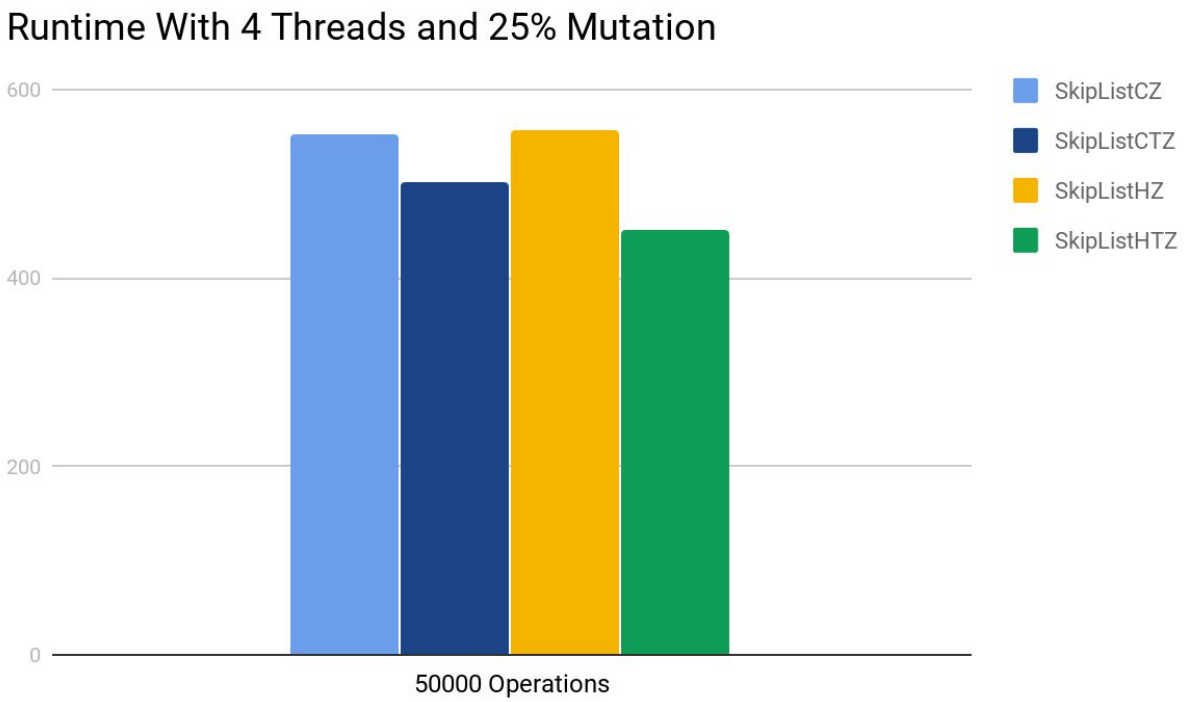

Figure 4. Measured SkipList code runtime with 50,000 operations.

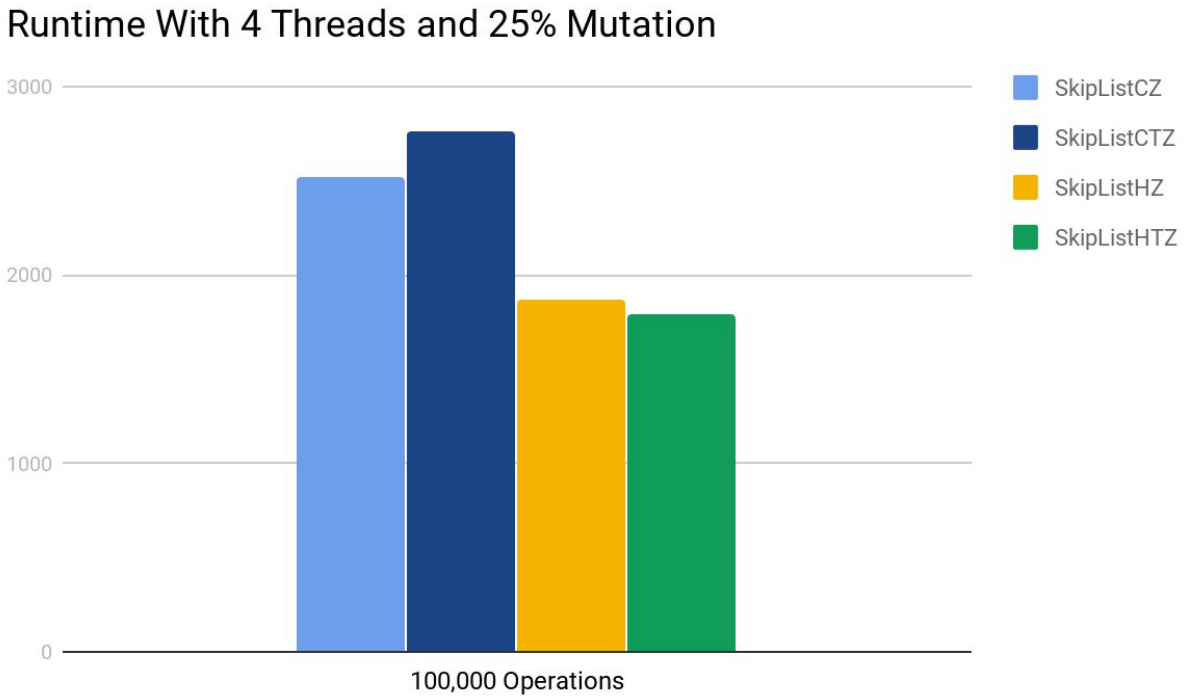

Figure 5. Measured SkipList code runtime with 100,000 operations.

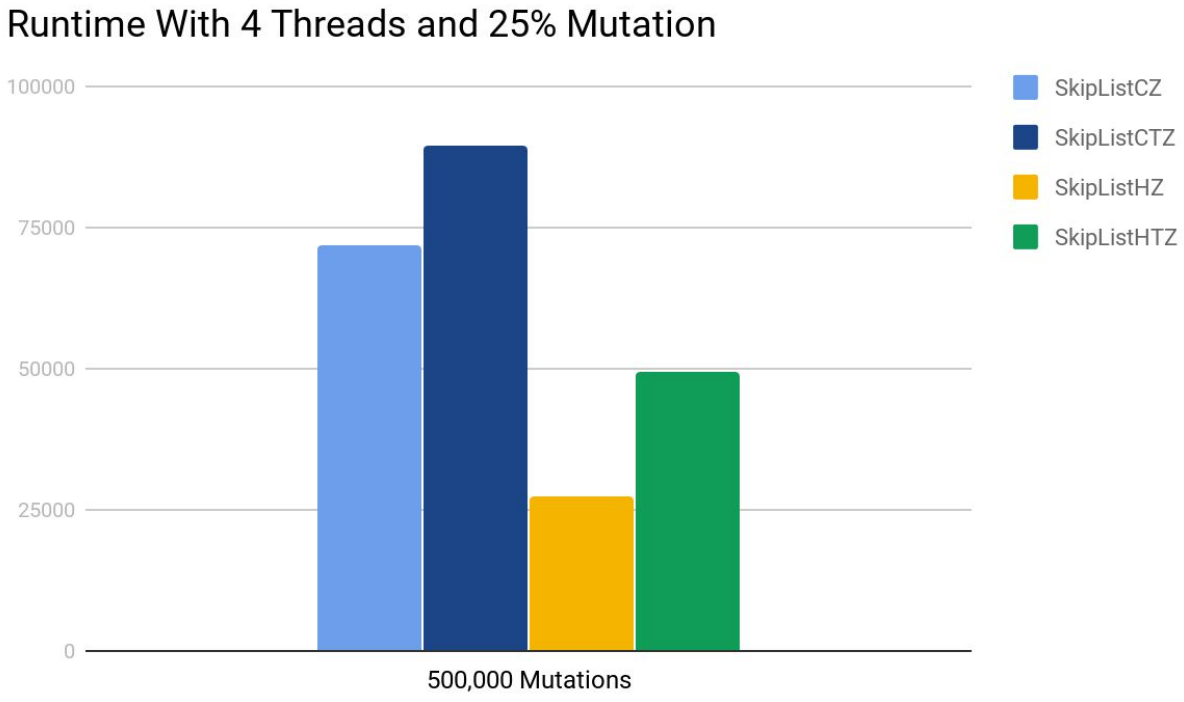

Figure 6. Measured SkipList code runtime with 500,000 operations.

At 50,000 operations, all four SkipLists performed comparably. However, as the number of operations increased, the hand-over-hand SkipLists performed significantly better than the coarse SkipList. In addition, each version of the code that utilized teleportation performed worse than its non-teleporting counterpart as the total number of operations increased despite performing better initially. Both versions of teleporting code also saw a decrease in teleportation distance as the number of operations increased. SkipListCTZ had an average teleportation distance of 175 nodes for the 50,000 operations, 145 nodes for 100,000 operations, and 16 nodes for 500,000 operations and the SkipListHTZ had an average teleportation distance of 13 nodes for 50,000 operations, 17 nodes for 100,000 operations, and 3 nodes for 500,000 operations.

IV. Discussion

The difference between the coarse and hand-over-hand SkipList measured performance was fairly expected: hand-over-hand locking benefits more from the addition of more threads than coarse locking. However, it was not expected for the coarse locking to ever do better than hand-over-hand locking, which appeared to be the case with a very small number of threads. A possible cause for this result is that the process of acquiring and releasing locks slightly slows down the process and hand-over-hand locking acquires a sizeable amount more locks than coarse locking. In addition, it appears that hand-over-hand locking seemed to have a linear increase in runtime when increasing the percentage of mutations, whereas the coarse locking seems to have a exponential increase in runtime.

It is also interesting to note that although the teleportation code typically had comparable runtimes as non-teleporting code, teleportation almost uniformly performed worse than non-teleportation in execution runtime. SkipListHTZ typically had very low teleportation

distances in general, typically between 18-30 nodes. SkipListHTZ also demonstrated decreasing success when the mutation percentage was increased, which is believed to be due to the fact that mutations take longer than contains function calls and locks are held longer, so more transactions encounter locks and have to abort. However, the distance SkipListHTZ would teleport increased as the percentage of mutations increased, due to the increased size of the list. In comparison, SkipListCTZ typically demonstrated very high teleportation distances, anywhere between 100 to 275 nodes. This trend is possibly explained both since SkipListCTZ does not have to acquire nearly as many locks, which allows it to only store the traversal portion of the transaction, and since only one thread can access the critical section, which prevents a thread from having to abort the transaction due to running into a lock. In addition, SkipListCTZ always demonstrated a high rate of successful transactions, which once again is believed to be due to the fact that only one thread is allowed into the critical section so no deadlocks are allowed. The teleportation code was expected to work significantly better at higher numbers of operations, but it seems that there is a golden window of number of operations where teleportation works best.

V. Conclusion

From this experiment, it appears there is no measurable improvement from the addition of teleportation to SkipLists for the purpose of list traversal; in fact, the lack of scalability seems to make adding teleportation to SkipLists a detriment. There may, however, be a significant improvement in efficiency by using teleportation for accommodating list mutations. The ability to change all of the pointers in atomic time could potentially allow the list to update all the pointers much faster, and may allow it to scale better with more operations.

VI. References

[1] “Lock (Computer Science).” *Wikipedia*, Wikimedia Foundation, 10 Apr. 2018, en.wikipedia.org/wiki/Lock_(computer_science).

[2] Muła, Wojciech. “File:Skip List.svg.” *File:Skip List.svg - Wikimedia Commons*, commons.wikimedia.org/w/index.php?curid=4871915.

[3]http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks-usage.pdf, page 8.

[4] “Restricted Transactional Memory Overview.” *Intel® Software*, Intel, 10 Oct. 2017, software.intel.com/en-us/node/524025.

[5] “Gethrtime.” *Synopsis - Man Pages Section 3: Basic Library Functions*, 1 Nov. 2011, docs.oracle.com/cd/E23824_01/html/821-1465/gethrtime-3c.html.

VII. Appendices

A. Crude Hazard-Management SkipList

```
SkipListCZ::SkipListCZ():
  LSentinel(LONG_MIN),
  RSentinel(LONG_MAX),
  hazardManager(2) {
  for(int i = 0; i < 5; i++) {
    LSentinel.next[i] = &RSentinel;
  }
};

void SkipListCZ::find(T v, SkipList::Node* predNodes[], ThreadLocal* threadLocal) {
  SkipList::Node* pred = &LSentinel;
  for (int level = LSentinel.height-1; level >=0; level--) {
    Node* curr = pred->next[level];
    while (curr->value < v) {
      pred = curr;
      curr = curr->next[level];
    }
    predNodes[level] = pred;
  }
};


bool SkipListCZ::add(T v, ThreadLocal* threadLocal) {
  lock();
  bool result;
  SkipList::Node* predNodes[LSentinel.height];
  find(v, predNodes, threadLocal);
  if(predNodes[0]->next[0]->value == v) {
    result = false;
  } else {
    result = true;
    SkipList::Node* node =  hazardManager.alloc(v, threadLocal);
    for (int layer = 0; layer < node->height; layer++) {
      node->next[layer] = predNodes[layer]->next[layer];
      predNodes[layer]->next[layer] = node;
    }
```

```
  }
  unlock();
  return result;
};


bool SkipListCZ::remove(T v, ThreadLocal* threadLocal) {
  lock();
  bool result;
  SkipList::Node* preds[LSentinel.height];
  find(v, preds, threadLocal);
  if (preds[0]->next[0]->value != v) {
    result = false;
  } else {
    SkipList::Node* nodeToDelete = preds[0]->next[0];
    for (int layer = 0; layer < nodeToDelete->height; layer++) {
      preds[layer]->next[layer] = nodeToDelete->next[layer];
    }
    hazardManager.retire(nodeToDelete, threadLocal);
    result = true;
  }
  unlock();
  return result;

};

bool SkipListCZ::contains(T v, ThreadLocal* threadLocal) {
  lock();
  bool result;
  SkipList::Node* preds[LSentinel.height];
  find(v, preds, threadLocal);
  result = (preds[0]->next[0]->value == v);
  unlock();
  return result;
};

void SkipListCZ::lock() {
  LSentinel.lock();
};

void SkipListCZ::unlock() {
  LSentinel.unlock();
};
```

B. Hand-Over-Hand Hazard-Management SkipList

```
SkipListHZ::SkipListHZ():
  LSentinel(LONG_MIN),
  RSentinel(LONG_MAX),
  hazardManager(2) {
  for(int i = 0; i < 5; i++) {
    LSentinel.next[i] = &RSentinel;
  }
};

void SkipListHZ::find(T v, SkipList::Node* preds[], ThreadLocal* threadLocal) {
```

```
  SkipList::Node* pred = &LSentinel;
  pred->lock();
  for (int level = LSentinel.height-1; level >=0; level--) {
    Node* curr = pred->next[level];
    preds[level] = pred;
    while (curr->value < v) {
      curr->lock();
      if (level == LSentinel.height-1 || pred != preds[level+1]) {
  pred->unlock();
      }
      pred = curr;
      curr = curr->next[level];
    }
  }
}

bool SkipListHZ::add(T v, ThreadLocal* threadLocal) {
  bool result;
  SkipList::Node* predNodes[LSentinel.height];
  find(v, predNodes, threadLocal);
  if (predNodes[0]->next[0]->value == v) {
    result = false;
  } else {
    SkipList::Node* node =  hazardManager.alloc(v, threadLocal);
    for (int layer = 0; layer < node->height; layer++) {
      node->next[layer] = predNodes[layer]->next[layer];
      predNodes[layer]->next[layer] = node;
    }
    result = true;
  }
  unlock(predNodes);
  return result;
};


bool SkipListHZ::remove(T v, ThreadLocal* threadLocal) {
  bool result;
  SkipList::Node* preds[LSentinel.height];
  find(v, preds, threadLocal);
  if (preds[0]->next[0]->value > v) {
    result = false;
  } else {
    SkipList::Node* nodeToDelete = preds[0]->next[0];
    nodeToDelete->lock();
    for (int layer = 0; layer < nodeToDelete->height; layer++) {
      preds[layer]->next[layer] = nodeToDelete->next[layer];
    }
    hazardManager.retire(nodeToDelete, threadLocal);
    result = true;
  }
  unlock(preds);
  return result;
};

bool SkipListHZ::contains(T v, ThreadLocal* threadLocal) {
  SkipList::Node* pred = &LSentinel;
  pred->lock();
  int distance = 0;
  for (int layer = LSentinel.height-1; layer >=0; layer--) {
    SkipList::Node* curr = pred->next[layer];
    while (curr->value < v) {
```

```
      distance++;
      curr->lock();
      pred->unlock();
      pred = curr;
      curr = pred->next[layer];
    }
    if (curr->value == v) {
      pred->unlock();
      return true;
    }
  }
  pred->unlock();
  return false;
};

void SkipListHZ::unlock(Node* nodes[], int start, int stop) {
  Node* lastUnlocked = NULL;
  for (int layer = start; layer < stop; layer++) { // back out
    if (nodes[layer] != lastUnlocked) {
      nodes[layer]->unlock();
      lastUnlocked = nodes[layer];
    }
  }
}
```

C. Crude Teleporting Hazard-Management SkipList

```
SkipListCTZ::SkipListCTZ():
  LSentinel(LONG_MIN),
  RSentinel(LONG_MAX),
  hazardManager(2) {
  for(int i = 0; i < 5; i++) {
    LSentinel.next[i] = &RSentinel;
  }
};

void SkipListCTZ::find(T v, SkipList::Node* predNodes[], ThreadLocal* threadLocal) {
  SkipList::Node* pred = &LSentinel;
  for (int level = LSentinel.height-1; level >=0; level--) {
    while (pred->next[level]->value < v) {
      pred = teleport(pred, v, threadLocal, level);
    }
    predNodes[level] = pred;
  }
};


bool SkipListCTZ::add(T v, ThreadLocal* threadLocal) {
  lock();
  bool result;
  SkipList::Node* predNodes[LSentinel.height];
  find(v, predNodes, threadLocal);
  if(predNodes[0]->next[0]->value == v) {
    result = false;
  } else {
    result = true;
    SkipList::Node* node =  hazardManager.alloc(v, threadLocal);
    for (int layer = 0; layer < node->height; layer++) {
```

```
      node->next[layer] = predNodes[layer]->next[layer];
      predNodes[layer]->next[layer] = node;
    }
  }
  unlock();
  return result;
};


bool SkipListCTZ::remove(T v, ThreadLocal* threadLocal) {
  lock();
  bool result;
  SkipList::Node* preds[LSentinel.height];
  find(v, preds, threadLocal);
  if (preds[0]->next[0]->value != v) {
    result = false;
  } else {
    SkipList::Node* nodeToDelete = preds[0]->next[0];
    for (int layer = 0; layer < nodeToDelete->height; layer++) {
      preds[layer]->next[layer] = nodeToDelete->next[layer];
    }
    hazardManager.retire(nodeToDelete, threadLocal);
    result = true;
  }
  unlock();
  return result;

};

bool SkipListCTZ::contains(T v, ThreadLocal* threadLocal) {
  lock();
  bool result;
  SkipList::Node* preds[LSentinel.height];
  find(v, preds, threadLocal);
  result = (preds[0]->next[0]->value == v);
  unlock();
  return result;
};

SkipListCTZ::Node* SkipListCTZ::teleport(Node* start, T v, ThreadLocal* threadLocal, int
level) {
  Node* pred = start;
  Node* curr = start->next[level];

  if (threadLocal->teleportLimit < 0) {
    threadLocal->teleportLimit = DEFAULT_TELEPORT_DISTANCE;
  }
  threadLocal->teleports++;
  while (true) {
    int distance = 0;
    unsigned int reason = Set::xbegin(threadLocal);
    if (reason == _XBEGIN_STARTED) {
      while (curr->value < v) {
        pred = curr;
        curr = curr->next[level];
        if (distance++ > threadLocal->teleportLimit) {
          break;
        }
      }
      _xend();
      threadLocal->tCommitted++;
```

```
      threadLocal->teleportDistance += distance;
      threadLocal->teleportLimit++;
      return pred;
    } else if (threadLocal->teleportLimit > 2) { // teleport less
      char* buff = new char[44];
      Set::parseAbort(reason, buff);
      threadLocal->teleportLimit /= 2;
    } else {      // non-transactional fallback
      threadLocal->fallback++;
      char* buff = new char[44];
      Set::parseAbort(reason, buff);
      threadLocal->teleportDistance++;
      threadLocal->teleportLimit = 2;
      if(curr->value < v) {
        return curr;
      }
      return pred;
    }
  }
};

void SkipListCTZ::lock() {
  LSentinel.lock();
};

void SkipListCTZ::unlock() {
  LSentinel.unlock();
};
```

D. Hand-Over-Hand Teleporting Hazard-Management SkipList

```
SkipListHTZ::SkipListHTZ():
  LSentinel(LONG_MIN),
  RSentinel(LONG_MAX),
  hazardManager(2) {
  for(int i = 0; i < 5; i++) {
    LSentinel.next[i] = &RSentinel;
  }
};

void SkipListHTZ::find(T v, SkipList::Node* preds[], ThreadLocal* threadLocal) {
  SkipList::Node* pred = &LSentinel;
  pred->lock();
  for (int level = LSentinel.height-1; level >=0; level--) {
    SkipList::Node* curr = teleport(pred, v, threadLocal, level);
    pred = curr;
    if(level == (LSentinel.height - 1)) {
      LSentinel.unlock();
    }
    while(pred->next[level]->value < v) {
      curr = teleport(pred, v, threadLocal, level);
      pred->unlock();
      pred = curr;
    }
    preds[level] = curr;
  }
};
```

```
bool SkipListHTZ::add(T v, ThreadLocal* threadLocal) {
  bool result;
  SkipList::Node* preds[LSentinel.height];
  find(v, preds, threadLocal);
  if(preds[0]->next[0]->value == v) {
    result = false;
  } else {
    result = true;
    SkipList::Node* node =  hazardManager.alloc(v, threadLocal);
    for (int layer = 0; layer < node->height; layer++) {
      node->next[layer] = preds[layer]->next[layer];
      preds[layer]->next[layer] = node;
    }
  }
  unlock(preds);
  return result;
};


bool SkipListHTZ::remove(T v, ThreadLocal* threadLocal) {
  bool result;
  SkipList::Node* preds[LSentinel.height];
  find(v, preds, threadLocal);
  if (preds[0]->next[0]->value != v) {
    result = false;
  } else {
    SkipList::Node* nodeToDelete = preds[0]->next[0];
    for (int layer = 0; layer < nodeToDelete->height; layer++) {
      preds[layer]->next[layer] = nodeToDelete->next[layer];
    }
    hazardManager.retire(nodeToDelete, threadLocal);
    result = true;
  }
  unlock(preds);
  return result;

};

bool SkipListHTZ::contains(T v, ThreadLocal* threadLocal) {
  bool result;
  SkipList::Node* preds[LSentinel.height];
  find(v, preds, threadLocal);
  result = (preds[0]->next[0]->value == v);
  unlock(preds);
  return result;
};

SkipListHTZ::Node* SkipListHTZ::teleport(Node* start, T v, ThreadLocal* threadLocal, int
level) {
  Node* pred = start;
  Node* curr = start->next[level];

  if (threadLocal->teleportLimit < 0) {
    threadLocal->teleportLimit = DEFAULT_TELEPORT_DISTANCE;
  }
  threadLocal->teleports++;
  while (true) {
    int distance = 0;
    unsigned int reason = Set::xbegin(threadLocal);
    if (reason == _XBEGIN_STARTED) {
      while (curr->value < v) {
```

```
        curr->lock();
        if(pred->value != start->value) {
          pred->unlock();
        }
        pred = curr;
        curr = curr->next[level];
        if (distance++ > threadLocal->teleportLimit) {
          break;
        }
      }
      _xend();
      threadLocal->tCommitted++;
      threadLocal->teleportDistance += distance;
      threadLocal->teleportLimit++;
      return pred;
    } else if (threadLocal->teleportLimit > 2) { // teleport less
      char* buff = new char[44];
      Set::parseAbort(reason, buff);
      threadLocal->teleportLimit /= 2;
    } else {      // non-transactional fallback
      threadLocal->fallback++;
      char* buff = new char[44];
      Set::parseAbort(reason, buff);
      threadLocal->teleportDistance++;
      threadLocal->teleportLimit = 2;
      if(curr->value < v) {
        curr->lock();
        return curr;
      }
      pred->lock();
      return pred;
    }
  }
};


void SkipListHTZ::unlock(SkipList::Node* nodes[]) {
  SkipList::Node* pred;
  for(int level= 0; level < LSentinel.height; level++) {
    SkipList::Node* curr = nodes[level];
    if(curr != pred) {
      curr->unlock();
    }
    pred = curr;
  }
};
```